Prototyping an aggregate programming framework with Scala/Akka

Project Report

Roberto Casadei
roberto.casadei12@studio.unibo.it

Complex Adaptive Software Systems Engineering course
Academic year 2014-2015, semester II
Alma Mater Studiorum – University of Bologna

Abstract. The Scala programming language and ecosystem provide developers with powerful tools for tackling complexity in software development, especially with regards to concurrency and distribution.
This project approaches the development of an aggregate programming framework with Scala and (possibly) the Akka actor framework.
The goal is to investigate whether these tools are adequate for such an effort and what key issues and challenges need to be tackled.

1 Introduction

Computers and software are important to us because of their ability to unburden us from the intricacies, the complication, and the complexity of our world. However, as we push forward our problems and our needs, the (software) systems themselves become more and more difficult to conceive, design, and develop. Our knowledge and tools should evolve to face the challenge of this increasing complexity. The question is: how can we effectively engineer complex, adaptive software systems?

One of the current trends in computing is represented by the field(s) of pervasive computing or Internet of Things. We may imagine a present/future where computation happens in the environment, driven by a dense network of (relatively simple) devices that intensively interact within certain physical constraints. How can we program such a multitude of devices in order to make interesting global behaviors predictably emerge out of local interactions and, at the same time, guarantee certain macro-level properties?

As computer scientists deal with such problems, we see ideas, abstractions, and paradigms arise. As these concepts effectively capture the essential complexity of the problems they address, we are left with the accidental complexity that may result from practical concerns or the lack of documentation or tools. It is not only a matter of productivity; support activities may also boost new insights by creating a suitable environment for scientific investigation.

2 Requirements

Briefly, our goal is to develop, in Scala/Akka, a framework to support *aggregate programming*, that is, the kind of programming paradigm promoted by MIT Proto[1] and often referred to as *spatial computing*.

We would like to be able to simulate systems that consist in a (possibly huge) number of elements (that we may simply call **devices** as often this model fits Internet of Things and pervasive computing applications) that interact with one another based on some notion of locality.

Step 1: rapid prototyping of a system for calculating the gradient

Here, the subgoal is to answer the following question: does Scala/Akka represent a suitable platform for building this kind of applications?

Step 2: extend the system to support distribution

Does Scala/Akka adequately support the development of distributed systems? We may want to set up a simulation where a few computational nodes run a subset of the devices each.

Step 3: turn the project to a (possibly, easy-to-use) framework

The primary goal remains the same: building a framework to facilitate the development and the simulation of spatial computing systems.

Step 4: investigate on the issue of code mobility

At this point, we ask: does Akka provide support for the shipping of code to remote components/actors? If not, how can we implement such a feature?

[1] `http://proto.bbn.com/commons/`

3 Software Project Description

3.1 Design forces

"The notion of force generalizes the kinds of criteria that software engineers use to justify designs and implementations." [2]
Some of the forces that impact on the framework architecture and the tools we use include:

- *Distribution* – this implies that many assumptions regarding latency, topology, or security are essentially false[3]
- *Asynchronicity*
- *Decentralization and local interactions*
- *Locality and spatial assumptions*
- *Interaction-based computation*
- *Dynamicity* axes
  - Behavior of a single device
  - System membership (devices leaving/entering the logical system boundaries)
  - Logical mobility (devices changing their spatial position)
  - Physical mobility (devices moving across computational nodes)?

3.2 An informal, conceptual overview

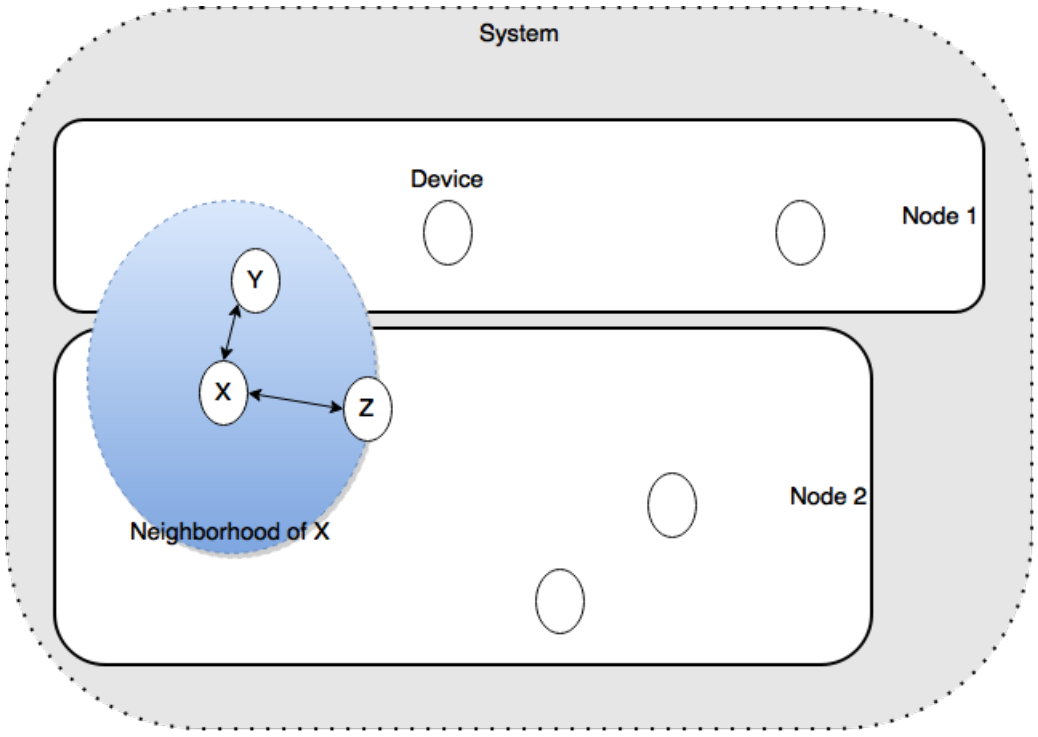

Fig. 1. Informal model of the system.

Notes

- Each device repeatedly runs its computation function and propagates its current state to its neighbors

[2] www.cs.unc.edu/~stotts/723/patterns/forces.html
[3] https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

- The computation of a device is a function of its state and a context given by its inputs (which comprise a neighbor-state map)
- Despite the simplicity of the computation function, we may observe global behaviors emerge out of the local interactions among the devices
- The system may be *distributed*, i.e., it may run on multiple physical machines (or nodes)
- The boundaries of the system may be "porous" in the sense that they may allow for devices to join or leave the system
- The notion of locality (that in turn defines the notion of *neighborhood* for a device) is a logical one (and is independent of the physical structure or the deployment of the system) but in the context of spatial computing it has a spatial connotation, so we can represent spatial locations as 3D points. If a device moves (i.e., changes its position), its neighborhood may change.

3.3 Scala project organization and dependencies

We use **sbt**[4] as our build automation tool.
The directory structure is shown below :

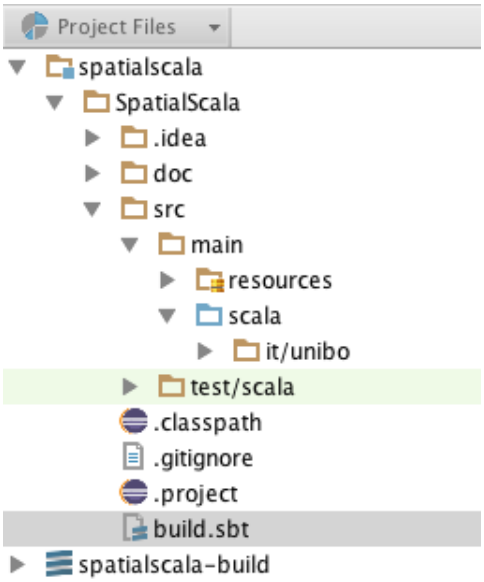

Fig. 2. Project files and directory hierarchy.

- Project source code and artifacts go under `src/main`
- Test source code and artifacts go under `src/test`
- The directory `.idea` is specific for the IntelliJ Idea IDE whereas `.classpath`, `.project` are relative to an Eclipse project.
- `.gitignore` includes the file patterns to be excluded from version control and suggests that the version control system is **Git**

`build.sbt` provides the build definition for this project and declares the project dependencies:

- **`akka-actor`**: basic Akka
- **`akka-remote`**: Akka Remote module
- **`scalatest`**: versatile testing framework supporting multiple testing styles
- `slf4j` and `scala-logging`: for logging purposes (outside Akka which comes with its logging facilities)
- `scopt`: for parsing command-line options

[4] SBT (Scala Build Tool) is the most widely used build tool in the Scala community.

3.4 High-level organization

The top-level package is **it.unibo.spala**. Then, the code is organized into 4 key subpackages:

- it.unibo.spala.**domain**: it includes the basic, reusable abstractions that pertains to spatial computing
- it.unibo.spala.**actors**: it includes the ingredients to build and simulate actor-based systems of interacting devices
- it.unibo.spala.**config**: it collects utilities and classes to support the configuration of Spala systems
- it.unibo.spala.**test**: it is intended to contain unit and integration tests

3.5 Basic concepts

The following elements can be found under the **it.unibo.spala.domain** package.

- A device **Dev[S]** has a name, a state of type S, and a behavior
- The behavior of a device **DevBehavior[S]** can be seen as a function of its state (of type S) and context, where the context gives access to the inputs and the neighbor map of the device

```
type MealyFunType[S, I] = (S, I) => S
type DevBehavior[S] = MealyFunType[S, DevBehaviorContext]

trait DevBehaviorContext {
  def getInput[T](name: String): Option[T]
  def getNeighborsStates[S]: Map[String, Option[S]]
}
```

- A space.**Space[E,P]** is an abstraction for a spatial container of elements: elements (of type E) can be retrieved and the space can be queried for the location (of type P) of an element
  - A **MutableSpace[E,P]** allows for the insertion, move and removal of elements
  - A **Basic3DSpace[E]** is a mutable space of elements of type E which are located by means of 3D points (x,y,z)
- A space.**DistanceStrategy[E,P,D]** defines how distance (of type D) and locality (neighborhood) are calculated for a given Space[E,P]
  - **EuclideanStrategy[E]** defines the euclidean distance (Double) on a 3D space

3.6 Key elements of a Spala actor system

The following elements can be found under the **it.unibo.spala.actors** package.

- device.**DevActor[S]** is the actor that represents a device (with state of type S). It internally dispatches a device.**DevComputationActor[S]** for executing the device behavior and may observe a number of device.**InputProviderActor**s which effectively work as sensors
- system.**SpaceManagerActor** is an abstract actor that represents a spatial container for actors. It is specialized by system.**SimpleSpaceManagerActor** which is a 3D space manager actor with an euclidean distance strategy
- env.**BasicGUIActor** observes the states of the device actors and provides a graphical representation of the evolution of the system along time
- DevActor and InputProviderActor are both **ObservableActor**s. Devices are observed by the GUI, whereas input providers (i.e., sensors) are observed by the devices themselves.

- Many actors (such as `DevActor`, `DevComputationActor`, `InputProviderActor`) implement the **`ActorLifecycleBehavior`** trait which factors out common features related to the lifecycle of components and the periodic execution of some task (according to a working interval)

ActorLifecycleBehavior

This trait, declared with self-type `Actor`, provides a common FSM-like behavior structure with template methods to be specialized.

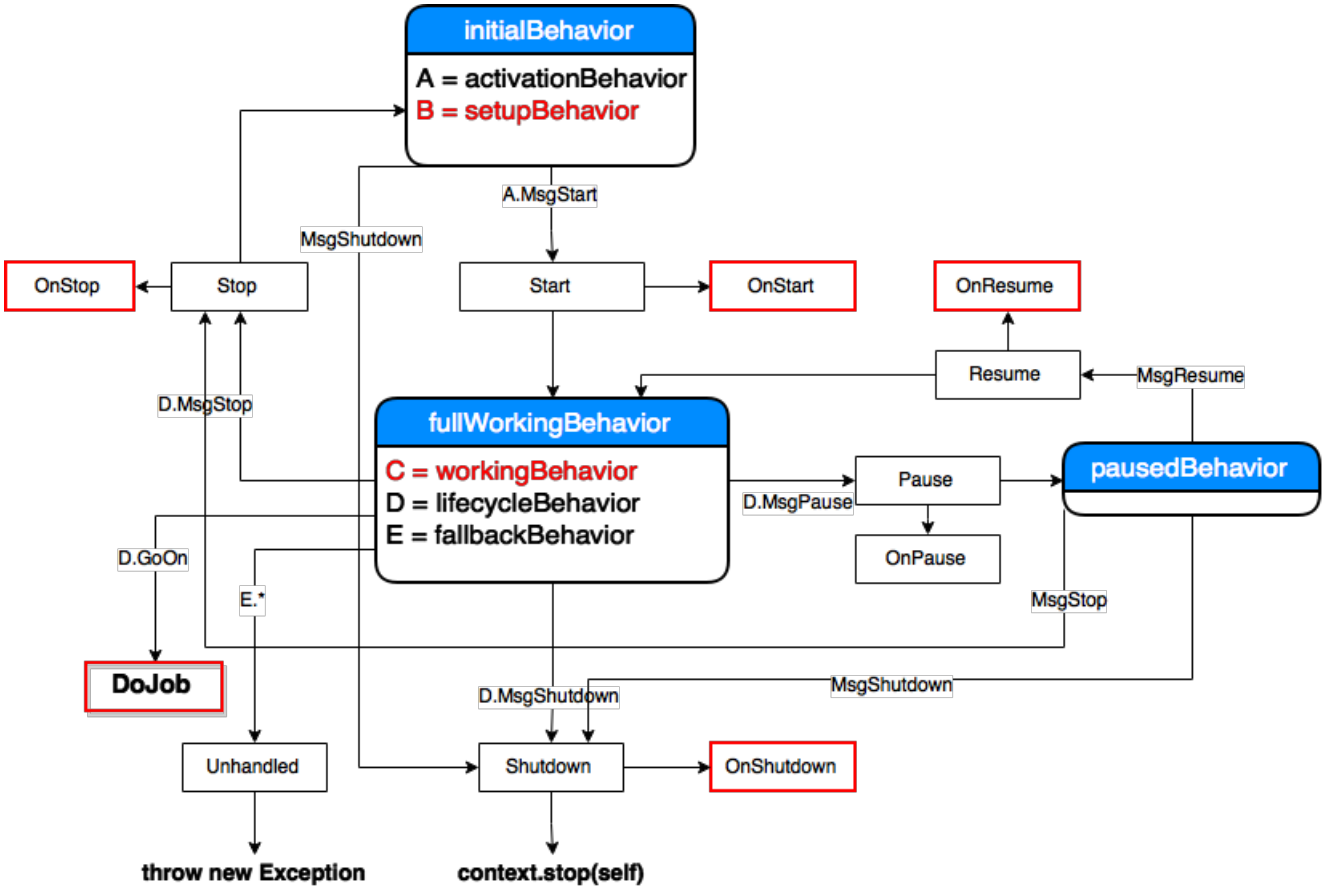

Fig. 3. The round boxes represent `Receive` behaviors. These may be composed of other behaviors; in that case, the priority is $A > B$. The squares represent method invocations. These methods are activated upon reception of certain messages, as defined by the current behavior. Notation $B.M$ means that message M is handled by the behavior B. The red-bordered squares are empty template methods that are provided and intended to be overridden in subclasses.

Note that such a structure has been conceived also to support periodic task execution; it suffices to call `ScheduleNextWorkingCycle(delay: FiniteDuration)` in `DoJob()` to auto-send a `GoOn` message after `delay`.

For example, `DevActor`'s `DoJob()` is responsible to propagate the device's state to its neighbors and notify observers, whereas `DevComputationActor`'s `DoJob()` is where the device computation function is actually executed.

3.7 Informal system architecture

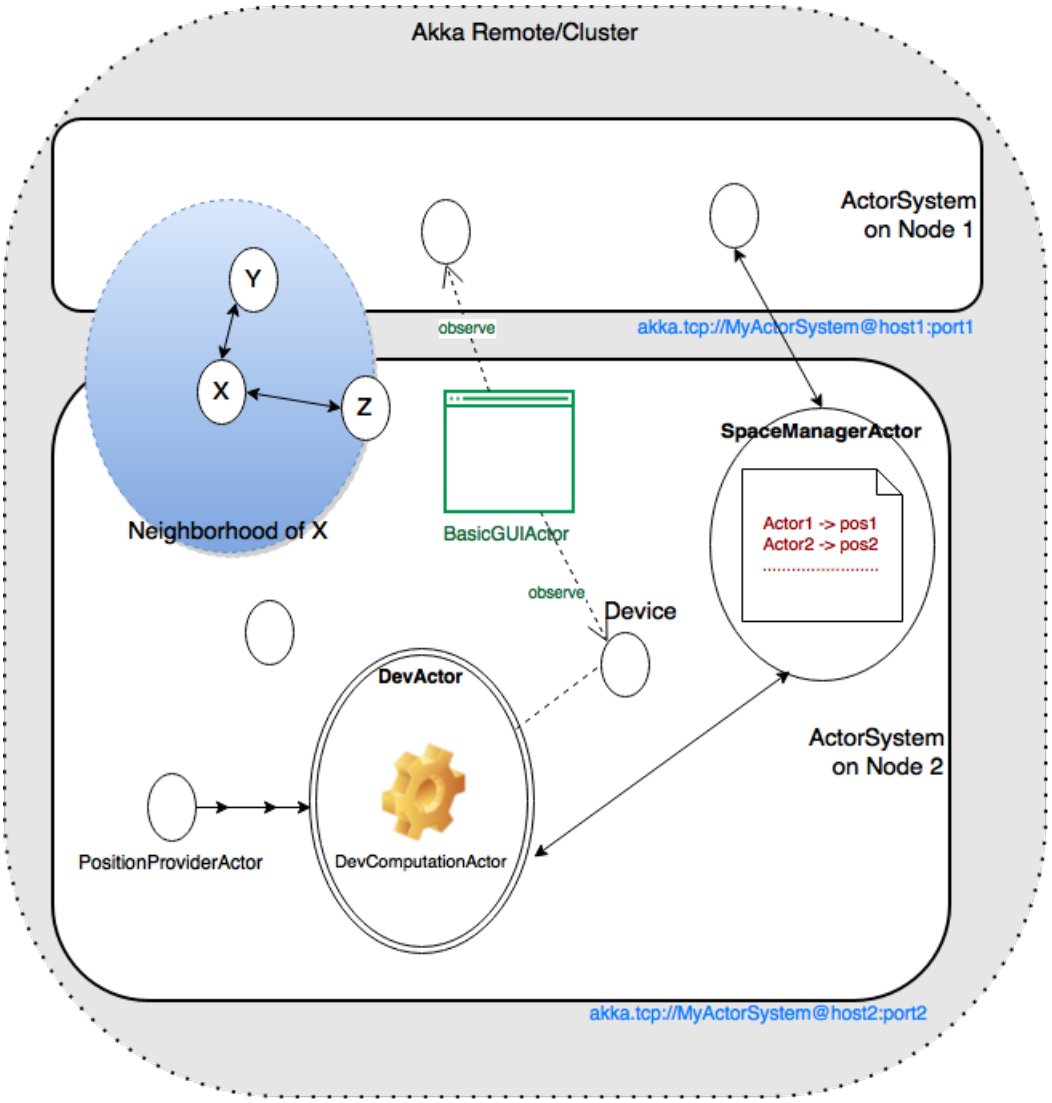

Fig. 4. Informal model of the actor system.

Notes

- A **device** is actually a mini-actor system that consists of three parts:
  1. the **device manager actor** – it interacts with the world external to the device, handles the sensors/inputs, keep tracks of its neighbors, and put the device behavior into execution by activating a computation actor
  2. the **computation actor** – it executes the device behavior and interacts with the device manager actor, e.g., to notify him with the newly computed state
  3. the collection of **input provider (sensor) actors**
- Device actors may register themselves to the **space manager actor** (which can be seen as their spatial container)
- The **GUI actor** observes the device actors: this means that, whenever a device updates its state, the GUI is notified
- **`Akka Remote`** is the Akka module that allows for *location transparency* – all you need to send a message to a remote actor is its `ActorRef`

3.8 Design models: dimension of structure

The following UML Class Diagram provides an excerpt of the structural design. Note that many important details have been omitted in favor of clarity and synthesis.

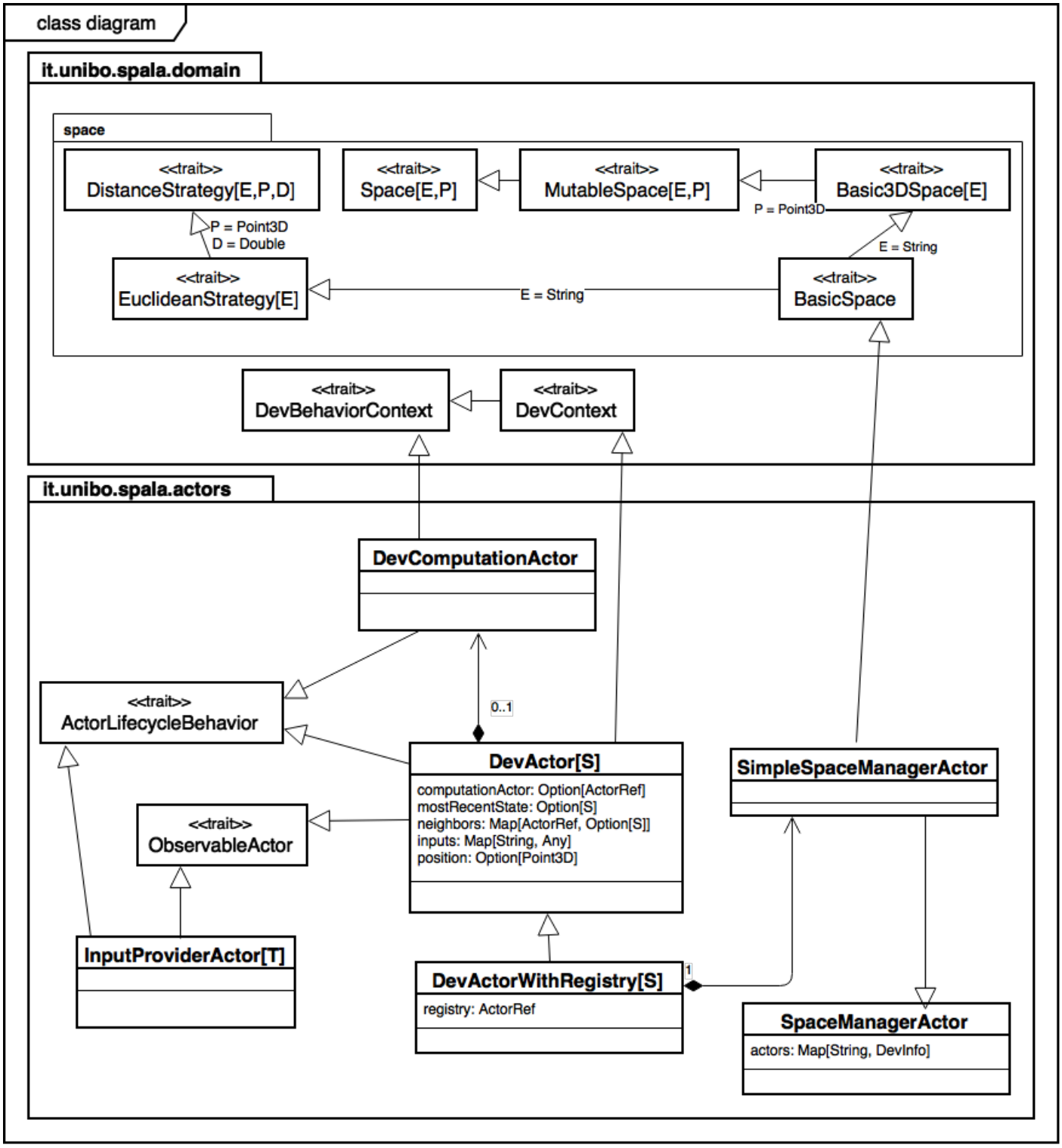

Fig. 5. Class diagram.

3.9 Design models: dimension of interaction

"Communication Diagram is a kind of UML interaction diagram which shows interactions between objects and/or parts (represented as lifelines) using sequenced messages in a free-form arrangement" [5].

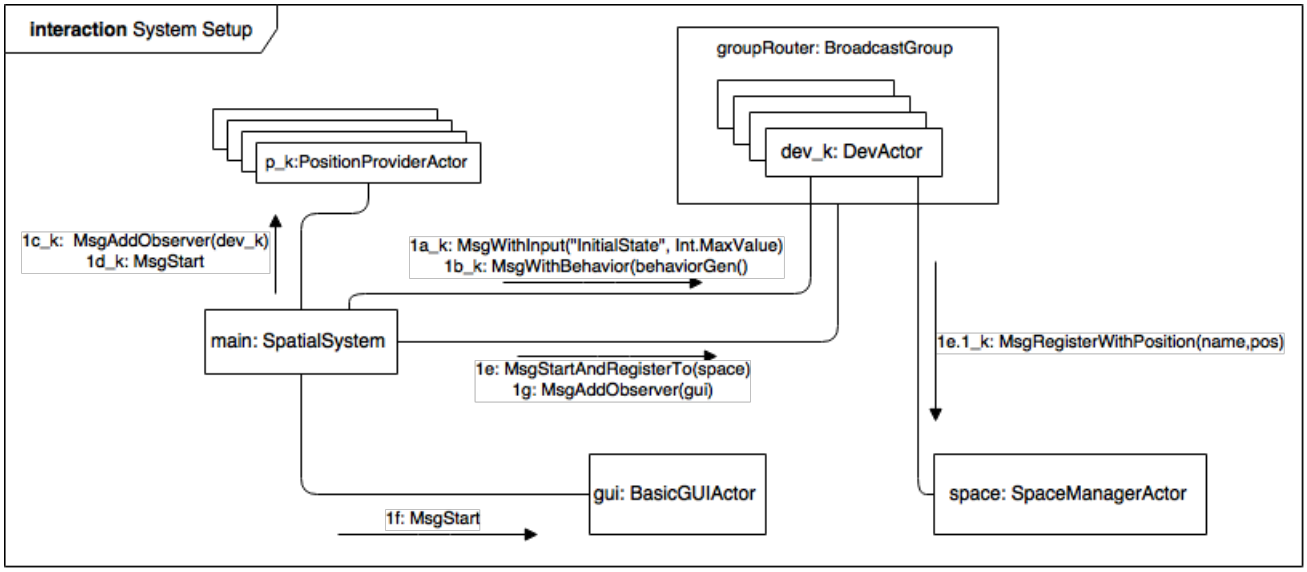

Fig. 6. Communication diagram of the setup phase of the system. The rectangles represent actors. The lines express association. The arrows represent messages. The numbers express the order in which messages are *sent*. Letters such as in 1.1*a* and 1.1*b* are used to express concurrent messages (here, within activation 1.1). Notation _k means that a message is dispatched to k recipients.

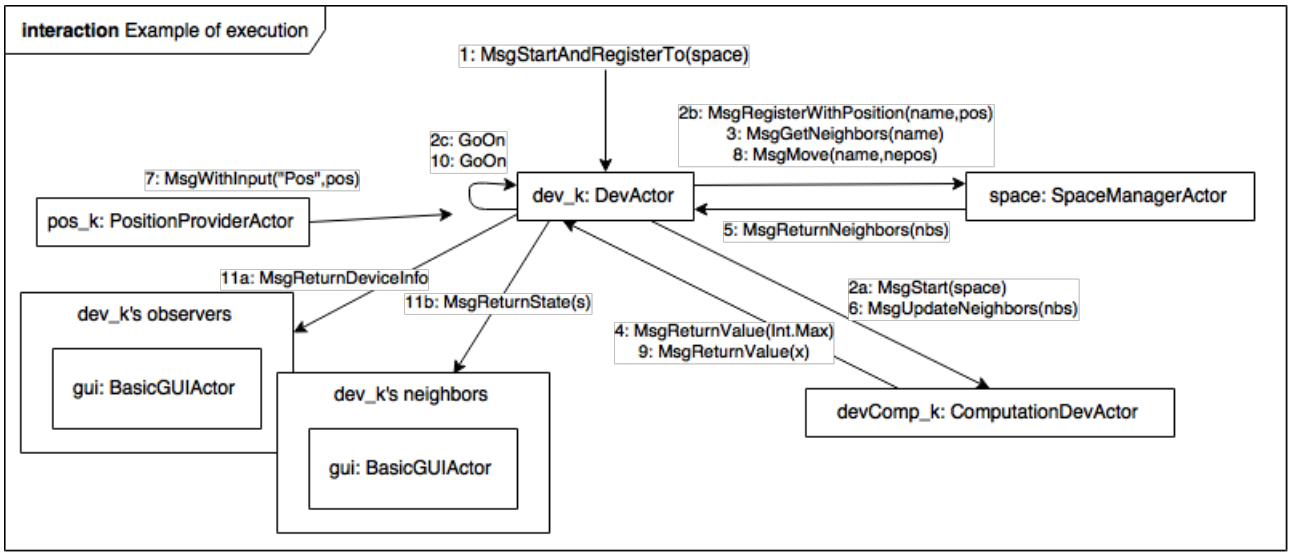

Fig. 7. Communication diagram showing an excerpt of the interactions for a single device actor. Note that, due to asynchronicity, several alternative message orderings are possible.

[5] http://www.uml-diagrams.org/communication-diagrams.html

4 Usage Guide

4.1 Source code

The source code can be downloaded from BitBucket. The repository URL is

`https://bitbucket.org/metaphori/spatialscala.git`

Download or clone the repository from your IDE or via `git clone`

```
~ $ git clone https://bitbucket.org/metaphori/spatialscala.git
```

4.2 Environment setup

Eclipse + Scala plugin (Scala IDE)

First, generate the Eclipse project files out of the SBT project.

```
~ $ cd spatialScala/SpatialScala
SpatialScala/ $ sbt
sbt> eclipse
```

Then, import the project with Eclipse via

`File | Import.. | General/Existing project into workspace`

IntelliJ IDEA + Scala plugin

From the Welcome window, select `Import Project`, navigate up the path `./spatialScala/SpatialScala` and then choose

`Import project from external model ⇒ SBT`

4.3 Running

Once your programming environment is ready, you may want to launch a program under `it.unibo.spala.programs` (such as `TestHighLevel`) or the tests under `it.unibo.spala.test` to verify that the framework is working.
The project is also shipped with a command-line main program that could be invoked from a shell:

```
$ sbt
sbt> runMain it.unibo.spala.programs.CmdLineProgram --help
<spatial program> 1.0
Usage: <spatial program> [start] [options]

  --prefix <prefix>
        Prefix for unique device names
  -g <value> | --gui <value>
        To launch a GUI
  -m <value> | --mobile <value>
        Dis/enables mobility for devices
  -d <N> | --proximity <N>
        Proximity (distance) threshold
  -n <N> | --numNodes <N>
        Number of nodes
  --grid <nrows>, <ncols>, <stepx>, <stepy>, <offsetx>, <offsety>
        Grid parameters
  -h <HOST> | --host <HOST>
        Host of deployment of the actor system
  -p <PORT> | --port <PORT>
        Port of deployment of the actor system
  --nodes <ADDR1>, ..., <ADDRN>
```

```
        Nodes that should be contacted on start
Command: start
start the system
  --help
        prints this usage text

sbt> runMain it.unibo.spala.programs.CmdLineProgram start -n 50
```

4.4 Creating a system: the easy way

When anything is working, you can use the framework to simulate various sorts of systems of interacting devices.

To easily configure and start a system, you can leverage the `SpatialSystem` class.

```
// Define the type of the state of the devices
type TState = Int

// Create the settings
val settings = Settings[TState](
    deviceSettings = DeviceSettings(
        initialDeviceState = Int.MaxValue,
        inputProvider = (k, name, ref, pos) =>
            List(InputData(InputNames.Source1, k==1))
    ),
    spatialSettings = SpatialSettings(proximityThreshold = 200)
)

// Define the behavior for the devices
// The following is actually already defined under DeviceBehaviors.
    GradientBehaviorFunction
val gradientFun: DevBehavior[Int] = (s: Int, ctx: DevBehaviorContext) =>
     {
    val states = ctx.getNeighborsStates[Int].values.map(_.getOrElse(Int.
    MaxValue))
    val isSrc = ctx.getInput[Boolean](InputNames.Source1).getOrElse(
    false)

    if (isSrc) 0
    else if (!states.isEmpty) {
        val min = states.min
        if(min<Int.MaxValue) min+1 else Int.MaxValue
    }
    else Int.MaxValue
}
// Define a function to generates the behavior for each device
// Note that it must have type () => DevBehavior[TState]
val behaviorGen = () => gradientFun

// Instantiate a SpatialSystem to setup and run the system
new SpatialSystem[TState](settings, behaviorGen)
```

Notes

- The settings field `inputProvider` is a function that generates, for each k-th device with name `name`, actor reference `ref` and position `pos`, a list of input values. In this example, the 1st device is given an input `InputData("isSrc", true)`, while the other devices are given `InputData("isSrc", false)`

Running this program will produce something similar as represented below.

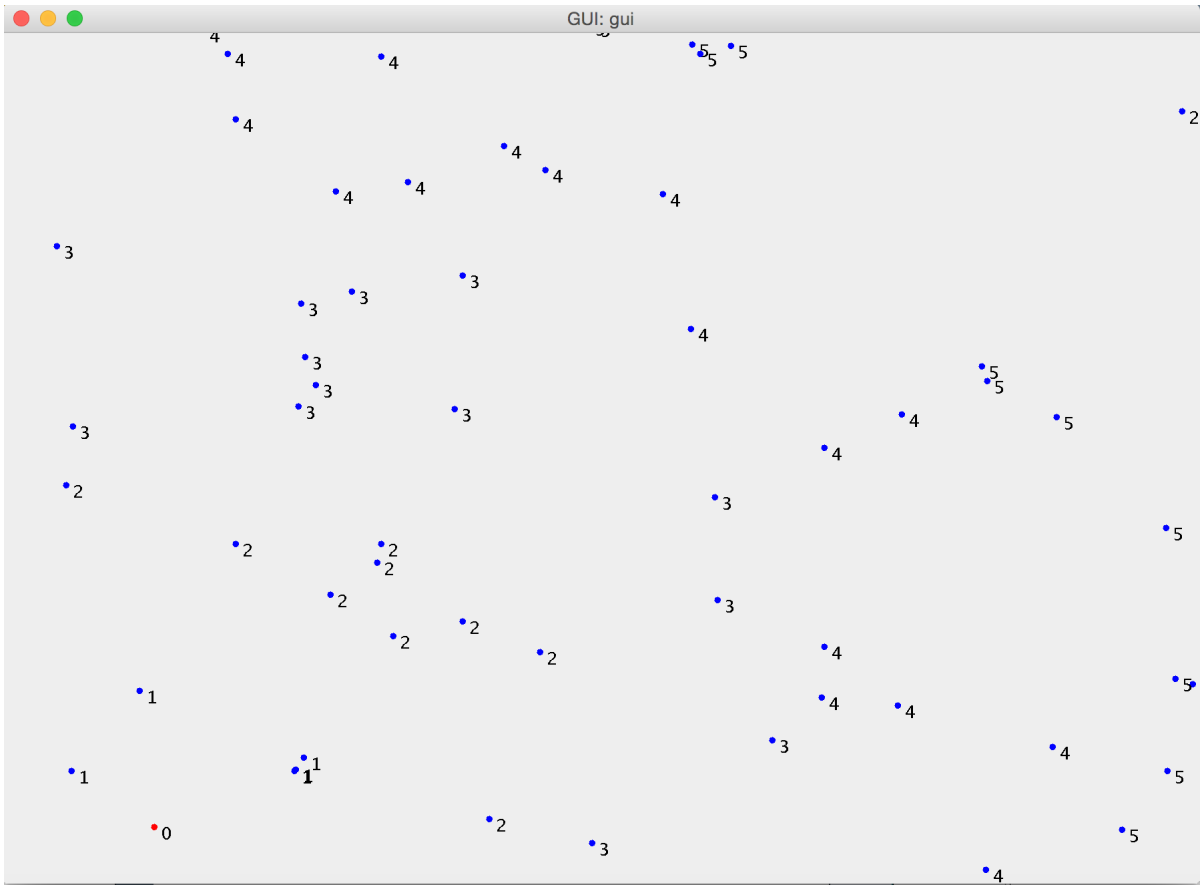

Fig. 8. Example: implementation of the gradient. The source device is the red point at the bottom-left corner.

Device behaviors

When it comes to define the behavior of the devices, you have at least three choices:

1. Reuse a behavior already defined as a member of the `it.unibo.spala.domain.`**`DeviceBehaviors`** object
2. Define your function of type **`DevBehavior[S]`** which is actually a type alias for `(s: S, ctx: DevBehaviorContext) => S`
3. Define a subclass of **`DBehavior[S]`** using one the following two approaches:

```
1  class ConstantBehavior(val value: Int) extends DBehavior[T] {
2      def apply(s: Int, ctx: DevBehaviorContext) = value
3  }
4
5  class GradientBehavior extends DBehavior[Int] {
6      does {
7          val states = nbsValuesOr[Int](Int.MaxValue)
8          val isSrc = inputOr[Boolean](InputNames.Source1, false)
9
10         if (isSrc) 0
11         else if (!states.isEmpty) {
12             val min = states.min
13             if(min<Int.MaxValue) min+1 else Int.MaxValue
14         }
15         else Int.MaxValue
16     }
17 }
```

The latter approach has the advantage to allow for coding the behavior from within an implicit context. However, such `DBehavior` instances are **NOT**

thread-safe, thus every device should run its own behavior object. This means that

```
val gradient = new GradientBehavior
val behaviorGenerator1 = () => gradient             // BAD!!
val behaviorGenerator2 = () => new GradientBehavior // GOOD
```

Settings for a spatial system

Please consult the definition of `Settings[T]` and related case classes to be acquainted with the configuration parameters and their defaults.

```
case class Settings[S](systemSettings: SystemSettings = SystemSettings(),
    deviceSettings: DeviceSettings[S] = DeviceSettings[S](),
    spatialSettings: SpatialSettings = SpatialSettings(),
    devDistributionSettings: DeviceDistributionSettings =
    DeviceDistributionSettings(),
    deploymentSettings: DeploymentSettings = DeploymentSettings())

case class SystemSettings(name: String = "MySpatialSystem",
    prefix: String = s"d${System.currentTimeMillis()}_",
    seed: Option[Long] = None,
    start: Boolean = true,
    executionStrategy: ExecutionStrategy = AsyncExecutionStrategy,
    gui: Boolean = true,
    console: Boolean = false,
    logLevel: String = AkkaConfigLogLevel.Info)

case class DeviceSettings[S](initialDeviceState: S = null.asInstanceOf[S],
    inputProvider: (Int, String, ActorRef, Point3D) => Seq[InputData[Any]] = (i,s
    ,a,p) => List(),
    nameProvider: (Int => String) = i => "Device"+i,
    workInterval: FiniteDuration = 100 millis,
    mobility: DeviceMobilitySettings = DeviceMobilitySettings())

case class DeviceMobilitySettings(canMove: Boolean = true,
    moveStep: Double = 20,
    moveProbability: Double = 0.1)

case class SpatialSettings(proximityThreshold: Double = 100)

case class DeviceDistributionSettings (n: Int = 100,
    shape: Symbol = 'random,
    shapeSettings: ShapeSettings = SimpleRandomSettings())

case class DeploymentSettings(kind: DeploymentKind = LocalDeployment,
    distrib: DistributedDeploymentSettings = DistributedDeploymentSettings(),
    local: LocalDeploymentSettings = LocalDeploymentSettings())

case class DistributedDeploymentSettings(host: String = "127.0.0.1",
    port: Int = 0,
    nodes: Seq[String] = List())

case class LocalDeploymentSettings()
```

Note that `DeviceDistributionSettings` is about the logical distribution of device in a space whereas `DistributedDeploymentSettings` concerns the physical, deployment-related system distribution.

Creating spatial configurations of devices

The **`ConfigHelper`** object contains a set of utility methods for the distribution of devices in space. Typically, these methods accepts a parameter to customize the procedure and a seed for randoms that could be used to enforce reproducibility.

- **`RandomLocations(SimpleRandomSettings)`**: randomly generates a list of 2D positions within a rectangle

```
case class SimpleRandomSettings(min: Double = 0,
                                max: Double = 1000) extends
    ShapeSettings
```

- **GridLocations(GridSettings)**: generates a list of positions along a grid, possibly with random deviations within a tolerance

```
case class GridSettings(nrows: Int = 10,
                        ncols: Int = 10,
                        stepx: Double = 100,
                        stepy: Double = 80,
                        tolerance: Double = 0,
                        offsetx: Double = 0,
                        offsety: Double = 0) extends ShapeSettings
```

4.5 Creating a system: the long way

You can take a look at the definition of `SpatialSystem` to see how a Spala actor system is built from scratch.
Abstracting from many configuration and setup details, the essential recipe is reported in the following listing.

```
package it.unibo.spala.programs

import akka.actor.{ActorSystem, Props}
import akka.routing.BroadcastGroup
import com.typesafe.config.ConfigFactory
import it.unibo.spala._
import it.unibo.spala.actors._
import it.unibo.spala.actors.device._
import it.unibo.spala.actors.env.BasicGUIActor
import it.unibo.spala.actors.system.SimpleSpaceManagerActor
import it.unibo.spala.domain._
import it.unibo.spala.math.Point2D

object SimpleProgram extends App {

  // We may need some randomness
  val rand = new scala.util.Random()

  // Type alias for the type of the devices' state
  type S = Int

  // Creation of the actor system
  val sys = ActorSystem(
    "MySystemName",
    ConfigFactory.defaultApplication())

  // Creation of the devices
  val devs = (1 to 10).map { n =>
    // Create actor for a device with Int state and get its reference
    sys.actorOf(
      Props(classOf[DevActorWithRegistry[S]]),
      "dev"+n)
  }

  // Definition of the behavior function
  val behaviorFun = (s: S, c: DevBehaviorContext) => {
    s+1 // Simply count forward..
  }

  // Then, for each device actor...
  devs.foreach { ref =>
    // Provide its initial state (BEFORE setting the behavior)
    ref ! MsgWithInput(InputData[S](InputNames.InitialState, rand.
    nextInt(9999)))
    // Provide its behavior
    ref ! MsgWithBehavior(behaviorFun)
    // Create a "GPS sensor" for its simulated position
    val positionInputActor = sys.actorOf(
      Props(classOf[PositionProviderActor],
        Point2D(rand.nextInt(999), rand.nextInt(999))))
    positionInputActor ! MsgAddObserver(ref)
    positionInputActor ! MsgStart
  }

```

```
54    // Create a broadcast router for the entire group of device actors
55    val broadcast = sys.actorOf(
56      BroadcastGroup(devs.map(_.path.toString)).props(),
57      "groupRouter")
58
59    // Creation of Space Manager, which keeps track of devices and their
        spatial positions
60    val registry = sys.actorOf(
61      Props(classOf[SimpleSpaceManagerActor]),
62      "registry")
63
64    // Create and start the GUI
65    val gui = sys.actorOf(
66      Props(classOf[BasicGUIActor], 800, 600),
67      "gui")
68    gui ! MsgStart
69
70    // Start device actors and register them to the space manager
71    broadcast ! MsgStartAndRegisterTo(registry)
72
73    // Make the GUI observe the devices
74    broadcast ! MsgAddObserver(gui)
75  }
```

../src/main/scala/it/unibo/spala/programs/SimpleProgram.scala

5 Further developments

First of all, there are a number of **project-related activities** to be done:

- Write unit and integration tests (see about Akka testing and TestKit[6])
- Refactoring (this project started with an exploratory inclination and well-defined time constraints)
- Error handling and fault-tolerance (see about Akka supervision strategies[7])
- The GUI should be generalized and extended to allow customized visualizations of the simulation

We also recognize the need for load and **performance testing**. Is the framework scalable? The current solution has an obvious centralization point, the `SpaceManager` actor which works as a registry and a space container. At the present time, many algorithms are a bit naive and could be largely improved.
Then, one important extension is the support of **code mobility**. In particular we would like to send to a remote device actor the code to be executed as its behavior.
Last but not least, the framework should evolve to provide the expressivity, in terms of style of programming and features, of **MIT Proto**. This style of programming (from global to local) is currently missing and is actually the ultimate reason of the project itself.

[6] `http://doc.akka.io/docs/akka/snapshot/scala/testing.html`
[7] `http://doc.akka.io/docs/akka/snapshot/scala/fault-tolerance.html`

6 Conclusion

The Scala/Akka platform seems to represent a good starting point for the development of a spatial computing framework, due to the presence of many traits able to contribute to the lowering of the abstraction gap, such as:

- Scala's integration of object-oriented and functional programming pardigms
- Akka's actor model of concurrency and infrastructure
- Java platform's richness in terms of libraries and documentation

In particular, the Actor model is promising since the systems we address are composed of a high number of distributed, autonomous, concurrent elements that interact asynchronously. For example, in a IoT scenario, it comes naturally to us to map any computational node with an actor and adopt a message-passing model of interaction in order to abstract out of the specific communication technology.
In this project, we have tested the fitness of the model and the easiness with which our systems can be brought to a distributed setting. To make a system distributed, it suffices to clear local system assumptions out, add a project dependency to `akka-remote` and deal with some configuration and deployment issue; application code does not have to change.
A first prototype has been developed in short time, despite some difficulties related to:

- learning curve for the Akka framework – from working details to the recovering of internal complexity that was hidden for the sake of simplicity of use
- learning curve for the Actor model – it is very clean for toy examples, but it does require careful design
- identification of the right abstractions – building a good model is not easy
- generalization of the found abstractions – there is an effort in enabling reuse and composability,

and this means that the abstraction gap is not excessive. In fact, many significant challenges related to concurrency and distribution do not need to be tackled as they have already been by Akka developers – these features are encapsulated at the level of the software infrastructure.
There is still work to do to get our spatial computing framework, but the basis is encouraging.

7 Attachment: on code mobility

One of the features we would like to support is code shipping to remote devices. In a simple scenario, a device could be started with no behavior, in a idle state; later on, some components or another device may determine the behavior that should be executed by the former device, and send it to him.
Supposing the behavior is defined as a lambda function (i.e., as an object of an anonymous inner class with a single method), the issue that we encounter in the Java platform is that the recipient actor must have access to the class definition in order to access the behavior object.
Given that Akka does not provide, at the infrastructure level, the way to send messages along with their class definition, we propose a solution based on custom class loaders.
Disclaimer: a major part of the solution has been found on the Internet.

7.1 A solution: shipping class byted + custom class loader

First of all, Akka allows to instantiate an `ActorSystem` with a custom classloader.

```
val ctxClassloader = Thread.currentThread().getContextClassLoader
val sys = ActorSystem(name,
                      config,
                      new CustomClassLoader(ctxClassloader))
```

Then, we prearrange an actor with the goal of receiving class definitions from remote systems.

```
class ClassLoaderActor extends Actor {
  override def receive: Actor.Receive = {
    case m: Map[String, Array[Byte]] => {
      m.foreach { case (clazz, classBytes) =>
        CustomClassLoaderRegistry.register(clazz, classBytes)
      }
    }
    case x => println("Ops, I got msg " + x + " but I dont know how to
    deal with it")
  }
}

sys.actorOf(Props[ClassLoaderActor], "classloader")
```

Where `CustomClassLoaderRegistry` just needs to keep track of the associations between class names and their definition byte arrays.

```
object CustomClassLoaderRegistry {
  var classesToRegister: Map[String, Array[Byte]] = Map()
  var registeredClasses: Map[String, Array[Byte]] = Map()

  def register(name: String, classBytes: Array[Byte]) {
    classesToRegister += (name -> classBytes)
  }

  def save(name: String): Unit ={
    registeredClasses += (name -> classesToRegister(name))
    classesToRegister -= name
  }
}
```

At this point, the two key missing pieces are the custom classloader itself and the utility method to get a class' bytecode.

Class serialization For a given class, the method should return a map of the class itself and its related classes to their definition codes.

```
object LoadClassBytes {
  def apply(clazz: Class[_]) : Map[String, Array[Byte]] = {
    val basePath : Path = Paths.get(clazz.getProtectionDomain.
    getCodeSource.getLocation.toURI)
    val relName = clazz.getName().replace('.', '/')
    val fullPath = basePath.resolve(relName)
    val fileName = fullPath.getFileName()
    val fileNameHead = (fileName.toString()).split("\\.")(0)
    val parentDir = fullPath.getParent()

    var res : Map[String, Array[Byte]] = Map()
    // find class file and class files of inner classes
    val ds = Files.newDirectoryStream(
      parentDir,
      new DirectoryStream.Filter[Path]{
        def accept(file: Path) : Boolean = {
          // Examples of a filename head: SomeClass, SomeClass$,
    SomeClass$$anonFun$3
          // Thus, base filename may include $ chars
          val fnameRegex = fileNameHead.replace("$","\\$")
          file.getFileName().toString().matches(fnameRegex+".*\\.class")
        }
      })
    try {
      val iter = ds.iterator()
      while (iter.hasNext()) {
        val p = iter.next()
        val classQualifiedName = ((basePath.relativize(p)).toString().
    split("\\.")(0).replace('/', '.'))
        res += (classQualifiedName -> Files.readAllBytes(p))
      }
    } finally {
      ds.close
    }
    res
  }
}
```

Custom class loader The custom class loader should work as by default and failback to the custom registry when classes are not found. By first looking at the registry we can accomplish a sort of class definition overriding.

```
class CustomClassLoader(parent: ClassLoader) extends URLClassLoader(
    Array[URL](), parent) {
  import CustomClassLoaderRegistry._

  override protected def findClass(name: String) : Class[_] = {
    var result = findLoadedClass(name);
    var toreg = classesToRegister.contains(name)
    if (result == null && !toreg) {
      try {
        result = findSystemClass(name);
      } catch {
        case e => () /* ignore */
      }
    }
    if (result == null && toreg) {
      try {
        val classBytes = classesToRegister(name)
        result = defineClass(name, classBytes, 0, classBytes.length);
        save(name)
      } catch {
        case e: Exception => {
          throw new ClassNotFoundException(name);
        }
      }
    }
    result;
  }
}
```

Testing the solution The receiver-side of the system just needs to create a `SpatialSystem`. When instantiating the `ActorSystem`, it will provide a `CustomClassLoader` instance); moreover, it will dispatch an `"/user/classloader"` actor for receiving class definitions.
The sender-side of the system will, first, send the class definition to its target system and then provide the behavior the a target actor:

```
val remoteBehavior = (s: Int, ctx: DevBehaviorContext) => {
    Console.println("This behavior will be executed on a remote actor")
    77 // just return a constant
}

val classRegistrationMsg = LoadClassBytes(remoteBehavior.getClass)

sys.actorSelection(remoteSystemPath + "/user/classloader")
    ! classRegistrationMsg

sys.actorSelection(remoteSystemPath + "/user/Device1")
    ! MsgWithBehavior(remoteBehavior)
```

Issues The main limitation is the *lack of transparency* to the users. It should be possible to arrange a part of the system to activate the shipping of the class definition upon need. When the custom class loader discovers that a class is not defined, it could request the class' bytecode from the remote actor system originating the request. The remote system has to provide an actor responsible of shipping class definitions. In this solution, one key issue is about how to associate a missing-class error with the situation of interest: a message involving objects of unknown class from a remote actor system. Moreover, how to save the message in order to be delivered at the end of the class-discovery process?
Another limitation is that the solution doesn't work with *closures*. Shipping class definitions is not enough, in some cases we also need a portion of the execution context.