ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Actor-Based Designs for Distributed Self-organisation Programming

This is the submitted version (pre peer-review, preprint) of the following publication:

Published Version:
Roberto Casadei, Ferruccio Damiani, Gianluca Torta, Mirko Viroli (2024). Actor-Based Designs for Distributed Self-organisation Programming. Berlin : Springer Nature [10.1007/978-3-031-51060-1_2].

Availability:
This version is available at: https://hdl.handle.net/11585/958108 since: 2024-02-15

Published:
DOI: http://doi.org/10.1007/978-3-031-51060-1_2

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2024

Actor-based Designs for Distributed Self-organisation Programming

Roberto Casadei[1[0000−0001−9149−949X]], Ferruccio Damiani[2[0000−0001−8109−1706]], Gianluca Torta[2[0000−0002−4276−7213]], and Mirko Viroli[1[0000−0003−2702−5702]]

[1] Alma Mater Studiorum–Università di Bologna, Cesena, Italy
{roby.casadei, mirko.viroli}@unibo.it
[2] Università degli Studi di Torino, Torino, Italy
{ferruccio.damiani,gianluca.torta}@unito.it

Abstract. Self-organisation and collective adaptation are highly desired features for several kinds of large-scale distributed systems including robotic swarms, computational ecosystems, wearable collectives, and Internet-of-Things systems. These kinds of distributed processes, addressing functional and non-functional aspects of complex socio-technical systems, can emerge in an engineered/controlled way from (re)active decentralised activity and interaction across all physical and logical system devices. In this work, we study how the Actors programming model can be adopted to support collective self-organising behaviours. Specifically, we analyse the features of the Actors model that are instrumental for implementing the adaptive coordination of large-scale systems, and discuss potential actor-based designs. Then, we discuss an incarnation of the approach in the aggregate computing paradigm, which stands as a comprehensive engineering approach for self-organisation. This is based on Akka, and can be fully programmed in the Scala programming language thanks to the ScaFi aggregate computing toolkit.

Keywords: Actors · Collective intelligence · Collective adaptive systems · Self-organisation · Programming models · Aggregate computing

1 Introduction

In the last decades, two key trends have been taking place in computer science and technology. First, more and more heterogeneous computing-enabled devices are being deployed into our environments, with larger scales and densities expected in the future, eventually creating enormous socio-technical ensembles. Secondly, there is an increasing need towards automation, demanding software systems to be more *autonomous* [21] (or *autonomic* [25]), and to exhibit so-called *self-* properties* [36] (e.g., self-managing, self-adaptive, self-repairing, etc.). These two trends together give rise to new potential applications and corresponding challenges, addressed through various approaches. In particular, a prominent *nature-inspired* [11] technique for the decentralised self-management

of large ensembles of computing devices is *self-organisation*, which is studied and implemented across different sub-fields of computer science [33,23,38,19]. A main classification of self-organisation engineering [12] is based on the distinction between *automatic* (i.e., based on learning and evolution) and *manual* approaches (where programmers use languages to express self-organisation programs—cf. *macroprogramming* [14]).

In this chapter, we focus on the latter approach and, in particular, we are interested in how *programming abstractions and paradigms* may support *self-organisation programming*. Specifically, we investigate how the *Actor model* can contribute to address the emergence of collective and self-organising behaviour. Indeed, self-organisation is generally related to peculiar aspects of actor systems, including reactivity, asynchrony, and locality. To do so, we develop actor-based solutions of well-known self-organising behaviours (*gradients* [5,30] and derived ones), and relate them with corresponding programs expressed in the *aggregate computing* paradigm [42] which is, currently and to the best of our knowledge, the most powerful and researched approach to self-organisation programming. What we find is that the plain Actor model has a relevant *abstraction gap* (distance between the problem and the solution), making it more suitable as a paradigm for the development of a middleware of a more high-level and declarative approach like aggregate computing, than as a solution for end-to-end design of self-organising behaviour. Still, research should be carried out to investigate what kinds of Actor extensions may help in the design and implementation of self-organisation, or what features of actors may improve aspects of aggregate computations (e.g., fine-grained scheduling of sub-computations).

The presentation is organised as follows. Section 2 provides background on self-organisation programming, reference examples of self-organising behaviour, and the Actor model (also through the Akka implementation [35]). Section 3 discusses actor-based designs of the self-healing gradient. Section 4 presents the actor-based design of the ScaFi aggregate computing middleware [17]. Finally, Section 5 provides a discussion and delineates directions for further research.

2 Background

In this section, we recall background information about self-organisation engineering and describe in detail two example self-adaptive algorithms (Section 2.1); then, we briefly recall the Actors model and its Akka implementation (Section 2.2).

2.1 Self-organisation and Collective Adaptive Systems

Self-organisation refers to the process whereby a system autonomously (i.e., without external control) seeks and sustains its order or structures [20]. It is often meant as a bottom-up decentralised process where macro-level structures and behaviours *emerge* from micro-level activities and interactions.

In modern cyber-physical systems such as the Internet of Things [4] and swarm robotics [12], self-organisation directly concerns the collective behaviour of large sets of computing and interacting devices. Engineering such systems is therefore a challenge of great practical importance, that can be addressed drawing from research areas such as *collective adaptive systems* [19], *macroprogramming* [14], *multi-agent systems* [45], and *aggregate computing* [42].

A main distinction in self-organisation engineering can be made between *automatic* approaches, whereby self-organising behaviour is learned (cf. multi-agent reinforcement learning [13,46]), evolved (cf. evolutionary robotics [39]), or synthesised [37]; and *manual* approaches [28], which are based on the definition of programs by programmers, e.g., in terms of control rules or designs involving patterns of information flow [44]. The manual approaches tend to differ based on the levels of *heterogeneity* and *scale*: small-scale heterogeneous systems can be programmed using *multi-agent programming* [10] or *choreographic* [29] approaches, whereas large-scale homogeneous systems are generally programmed using *macroprogramming* [14] approaches, such as *ensemble computing* [31], or *aggregate computing* [42] approaches. Note that *hybrid* automatic/manual approaches also exist—cf. approaches where program sketches are filled with automatically generated/searched behaviours [2].

In this chapter, we focus on *manual approaches for programming large homogeneous systems*. In particular, this activity can be supported by suitable *programming abstractions* supporting declarative specifications of collective behaviours. Examples of abstractions include first-class *ensembles* [31] or collective data structures like *computational fields* [27,42], In the following, we will focus on the computational field abstraction, offered by *Aggregate Computing (AC)*.

AC systems consist of a (possibly large) number of computational devices, connected in a network, and all operating at asynchronous *rounds* of execution, each round consisting of *sense–compute–act* steps, where the compute step involves the evaluation of an aggregate program against the currently sensed contextual information. An output or state of a whole or part of a distributed system can then be represented as a *field* of values computed by all the device constituting its domain. For instance, the movement of a swarm may be described by a field of velocity vectors; or, the temperature in a room may be denoted by the field of temperature readings of all the sensors there. The computational field is the fundamental abstraction for AC, and programming AC systems roughly means describing how such fields are manipulated in space-time. The essence of the programming model is captured by a minimal core language called the *field calculus* (FC) [6], which provides a set of functional constructs for handling the stateful evolution of fields and neighbour-based communications. A device can only directly communicate (in broadcast) with its *neighbours*, as defined by an application-specific logical or physical (ad-hoc) proximity relation. In each device, a round of computations consists mainly of three steps: (i) creation/update of the execution context, consisting of previous device state, the most recent messages received from neighbours, and values sampled from local sensors; (ii) local execution of the aggregate program, which produces a logically single result

(*output*); (iii) broadcast of part of the output to all the neighbours (this part is called the *export*), and possible activation of the actuators on the basis of the provided output.

Reference Example #1: Self-Healing Gradient As a first, simple example of a self-organising computation, we consider the *gradient*, namely the self-healing field of minimum path distances from any node to a source node. A simple implementation is based on the distributed Bellman-Ford algorithm, to be executed by all the devices repeatedly in rounds (where the rounds serve to integrate and propagate up-to-date information):

$$D(\delta, src) := \begin{cases} 0 & \text{if } src(\delta) \\ \min\{D(\delta') + d(\delta, \delta') : \ \delta' \in N(\delta)\} & \text{otherwise.} \end{cases} \tag{1}$$

The algorithm estimates the minimum distance of a device from a *source* device (i.e., a device where predicate $src()$ is true). We assume that function $d()$ returns the *current* distance between two devices, and $N()$ returns the set of current neighbours of a device. At each round, a device δ which is not a source, estimates $D(\delta)$ by considering the set of distances $D(\delta')+d(\delta, \delta')$ that separate it from the source through each one of its neighbours δ', and taking the minimum of those.

Two observations are in order: first of all, it is easy to see that, if the network is stable (i.e., devices do not crash, do not move, and do not join/leave the network), the algorithm actually converges to the correct value in each device δ. Secondly, after *any* of the above changes happen, if the network stabilises again for enough time, the values in each device δ are updated with the new correct values. In other words, the algorithm is *self-stabilising* [41]. Even if simple, the algorithm is both *collective*, i.e., fully distributed among the participating devices, and *adaptive*, i.e., resilient to the relevant changes in the system and the environment.

The following code is the implementation of the algorithm in the ScaFi language [17], a Scala-based implementation of Field Calculus.

```
def gradient(source: Boolean): Double =
  rep(Double.PositiveInfinity) { dist =>
    mux(source){ 0.0 } { minHood(nbr{dist} + nbrRange()) }
  }
```

The `rep` construct propagates the computed gradient value between rounds (in this case, the value computed by `mux`). The `nbr` construct can be thought of as returning the neighbouring field with the last values of `dist` received from the neighbours. Finally, `nbrRange()` returns a neighbouring field with the distance estimates to the neighbours, and `minHood()` returns the minimum value of a field. Also, note that the `gradient` function directly takes a Boolean value indicating whether the current device δ is a source, and that the δ parameter is not passed explicitly to `gradient` (since the program is evaluated locally to each device, there is always an implicit current device).

Reference Example #2: Self-Healing Channel A more complex example of self-organising computation is the *self-healing channel*, namely the construction of a path of devices across the network connecting a source device to a destination device, where the fact of belonging or not to the channel can be denoted by a local Boolean output (i.e., the channel consists of all the devices of the network the output `true`). Since this can be implemented on top of (a generalisation of) gradients, it is instrumental to convey the idea of *compositionality* of self-organising behaviours.

Taking inspiration from [41], let us generalize the D function to a higher-order operator G as follows:

$$G(\delta, src, ini, acc, met)$$

where src is the source of the field to be constructed, ini is the input value of the field to be considered, acc is the function expressing how to accumulate values starting from the source outwards (i.e., how to integrate local values ini to the accumulated value taken from the neighbour minimising the gradient in the neighbourhood), and met the metric of the distance between two devices. The self-healing gradient above can then be expressed as:

$$D(\delta, src) := G(\delta, src, 0, \lambda x.(x + d), d)$$

where we have used the lambda calculus notation for defining the acc function. We exploit the G operator to define another function, broadcast (B):

$$B(\delta, src, val) := G(\delta, src, val, \lambda x.x, d)$$

Assuming a single source device δ_{SRC} for which $src(\delta_{SRC})$ is true, this function broadcasts a value val defined in δ_{SRC} unaltered (thanks to using the identity function for acc) to all the other nodes, at increasing distances.

Let us consider the problem of establishing a robust communication channel between a source device δ_{SRC} and a destination/target device δ_{TRG} in a network with proximity-based communication. Starting from the B and D functions defined above, we can first of all define a function which broadcasts everywhere the distance between a source and a target, where src and trg are predicates that are true, respectively in the source and target of the communication channel:

$$BTW(\delta, src, trg) := B(\delta, src, D(\delta, trg))$$

Then, we can define a function that, for every device δ, is true iff δ belongs to the communication channel between the source and target devices:

$$CH(\delta, src, trg, w) := D(\delta, src) + D(\delta, trg) \leq BTW(\delta, src, trg) + w$$

Note that a device belongs to the channel iff it falls within an ellipse whose foci are the source and target devices. The w parameter determines the "stretch" of the ellipse, which reduces to a linear path in case $w = 0$. In particular, to determine which devices are part of the channel between δ_{SRC} and δ_{TRG}, we execute in every node:

$$CH(\delta, \lambda x.(x == \delta_{SRC}), \lambda x.(x == \delta_{DST}), w)$$

The following code is the implementation of the algorithm in the ScaFi language.

```
def broadcast[V:OB](source: Boolean, init: V): V =
  G[V](source, init, x=>x, nbrRange())

def distanceTo(source: Boolean): Double =
  G[Double](source, 0, _ + nbrRange(), nbrRange())

def distBetween(source: Boolean, target: Boolean): Double =
  broadcast(source, distanceTo(target))

def isSource = sense[Boolean]("source")
def isTarget = sense[Boolean]("target")

def channel(src: Boolean, dest: Boolean, width: Double) =
  distanceTo(src) + distanceTo(dest) <=
  distBetween(src, dest) + width

channel(isSource, isTarget)
```

Note that the predicates *src* and *trg* are replaced by virtual sensors that return the appropriate Boolean values.

2.2 The Actors Programming Model

The *Actor model* [24,1,26] puts *actors* at the core of the design and implementation of distributed systems. Actors are *reactive* agents that communicate with each other through *asynchronous message passing* (i.e., no shared memory is allowed).

It is worth noting that each message is directed to a specific actor through a *target address*, and that a mailbox system buffers messages until they are processed by their target actors. The actors in the distributed system execute in parallel. In particular, each actor iteratively and asynchronously processes the messages in its mailbox received from the other actors.

The fundamental part of the behaviour of an actor is specified in terms of how it handles incoming messages. In response to a message, an actor can:

- perform local computations;
- send messages to other actors;
- create new actors;
- choose the behaviour for handling the next message.

Handling of multiple messages is not interleaved or, analogously, handling of a single message is atomic.

Then, the Actor model can be formalised and implemented in different ways, possibly bringing in peculiar extensions. An example of implementation is provided by the *Akka toolkit* [35], whose user interface is briefly described in the following.

2.3 The Akka Toolkit: a Short Primer

We briefly illustrate the user *Application Program Interface (API)* of *Akka* [35], focussing on the *Akka Typed* version, which will be useful to understand the code provided in Section 3.

Actor behaviour Actor behaviour is dynamically represented through values of type `Behavior[M]`, which encapsulate the logic for handling messages of type `M`. So, an actor behaviour can be defined by extending `AbstractBehavior[M]` and overriding method `onMessage` (*OOP-style*), or by functions yielding a `Behavior[M]` (*functional style*). The Akka API provides a factory object `Behaviors` for specifying behaviours as functions mapping messages to the next behaviour, e.g., using pattern matching. Actors can be addressed through a reference of type `ActorRef[T]`: e.g., given a reference `r`, instruction `r ! m` denotes the sending of a message `m` of type `T` to the actor denoted by reference `r`.

Actor systems An actor system is created by instantiating an `ActorSystem[T]` with the `Behavior[T]` of the top-level actor; such a top-level actor would be responsible for *spawning* new actors by calling `ActorContext[T].spawn(behavior)`. Indeed, actor systems consist of a *hierarchy* of actors (enabling *supervision*), where each actor has a position in this hierarchy that can be denoted by a *path* of *actor names*, starting from the *top-level actor* `/user` (for user – i.e., non-system-level – actors): e.g., `/user/a/b` is the path of actor `b` which is a child of `a` (which is in turn a child of the top-level actor).

3 Actor-based Designs for Aggregate Computations

In this section, we discuss possible actor-based designs for building the paradigmatic self-organising behaviours covered in Section 2.1. The produced source code has been made available at a public repository[3] with a permissive licence, equipped with the build infrastructure for simple execution.

3.1 A Naive Actor-based Implementation of the Self-Healing Gradient Example

Figure 1 shows a possible implementation of the self-healing gradient within the Akka framework. This version is deliberately naive, and serves mainly as a baseline that will be refined in the next sections.

The application contains a single type of actor named `Device`. The actor defines a behaviour that matches several types of messages. The code executed to handle a `Round` serves as the initiation of a round of computation (cf. the aggregate computing execution model—see Section 2.1). More specifically:

- for each neighbour `nbr`, it requests the current value of the position (`GetPosition`) and of the gradient (`QueryGradient`)
- a timer is set to expire in one second and send a `ComputeGradient` message to the actor itself.

[3] `https://github.com/metaphori/experiment-actor-design-selforg`

```
object Device {
  def apply(src: Boolean,
            g: Double,
            nbrs: Map[ActorRef[Msg],Long],
            distances: Map[ActorRef[Msg],Double],
            nbrGs: Map[ActorRef[Msg],Double],
            pos: Point3D = Point3D(0,0,0)): Behavior[Msg] = Behaviors.setup { ctx =>
    val getPositionAdapter: ActorRef[NbrPos] =
      ctx.messageAdapter(m => SetDistance(m.pos.distance(pos), m.nbr))
    val getGradientAdapter: ActorRef[NbrGradient] =
      ctx.messageAdapter(m => SetNeighbourGradient(m.g, m.nbr))

    Behaviors.withTimers { timers => Behaviors.receive { case (ctx,msg) => msg match {
      case SetSource(s) =>
        Device(s, 0, nbrs, distances, nbrGs, pos)
      case AddNeighbour(nbr) =>
        Device(src, g, nbrs + (nbr -> currTime()), distances, nbrGs, pos)
      case RemoveNeighbour(nbr) =>
        Device(src, g, nbrs - nbr, distances, nbrGs, pos)
      case SetPosition(p) =>
        Device(src, g, nbrs, distances, nbrGs, p)
      case GetPosition(replyTo) =>
        replyTo ! NbrPos(pos, ctx.self)
        Behaviors.same
      case SetDistance(d, from) =>
        Device(src, g, nbrs + (from -> currTime()), distances + (from -> d), nbrGs, pos)
      case ComputeGradient =>
        val newNbrGradients = nbrGs + (ctx.self -> g)
        val disalignedNbrs = nbrs.filter(nbr => currTime() -
          nbrs.getOrElse(nbr._1, Long.MinValue) > RETENTION_TIME).keySet
        val alignedNbrGradients = newNbrG -- disalignedNbrs
        val alignedDistances = distances -- disalignedNbrs
        timers.startSingleTimer(Round, 1.second)
        if(src){
          Device(src, 0, nbrs, distances, newNbrG, pos)
        } else {
          val updatedG = (alignedNbrGradients - ctx.self).map(n => n -> (n._2 +
            alignedDistances.get(n._1).getOrElse(Double.PositiveInfinity))
          ).values.minOption.getOrElse(Double.PositiveInfinity)
          Device(src, updatedG, nbrs, distances, nbrGs + (ctx.self -> updatedG), pos)
        }
      case QueryGradient(replyTo) =>
        replyTo ! NbrGradient(g, ctx.self)
        Behaviors.same
      case SetNeighbourGradient(d, from) =>
        Device(src, g, nbrs + (from -> currTime()), distances, nbrGs + (from -> d), pos)
      case Round =>
        nbrs.keySet.foreach(nbr => {
          nbr ! GetPosition(getPositionAdapter) // query nbr for nbrsensors
          nbr ! QueryGradient(getGradientAdapter) // query nbr for app data
        })
        timers.startSingleTimer(ComputeGradient, 1.seconds)
        Behaviors.same
      case Stop => Behaviors.stopped
  } } } }
}
```

Fig. 1. A naive Akka implementation where a single actor encapsulates all the concerns.

The neighbour actors would reply to the `GetPosition` and `QueryGradient` requests, and the current actor stores the retrieved information in its state (specifically, in the `distances` and `nbrGs` maps). When the `QueryGradient` is received, further operations are performed to complete the round of computation:

- neighbours whose latest messages are too old (i.e., expired) are discarded (e.g., in order to become aware of device failing or quitting the system);
- a timer is set to expire in one second and send a `Round` message to the actor itself (i.e., to initiate the next round and possibly detect new information from the environment);
- if the actor is a `src` of the gradient computation, it just propagates its behaviour with the gradient set to constant `g = 0`;
- otherwise, the gradient is updated to the new value `updatedG` computed from the information retrieved from the neighbours, according to the logic of the gradient implementation illustrated in Section 2.1.

3.2 An Improved Design

The naive design of the previous section has several issues. The main issue is that the `Device` actor is not *reusable* but rather specific to the computation at hand: this is witnessed by application-specific messages (e.g., `ComputeGradient` and `SetNeighbourGradient`). Another issue is that the `Device` encapsulates all the concerns, including e.g. the scheduling concern (cf. the use of `timers` to schedule rounds and computations).

In Figure 2, an improved design is presented. It is also coded with a different style: the OOP style, instead of the functional style as in Figure 1, which is mainly a matter of taste, and in this case is more suitable to avoid encoding state into a large parameter list. In particular, the `DeviceActor` is an abstract class: to be implemented, the abstract `compute` method has to be defined (cf. the *Template Method* design pattern [22]). Additionally, the responsibility of scheduling has been moved outside of the actor: it will compute reactivity upon reception of a `Compute` message; it is straightforward to define a scheduler actor that keeps the references of the device(s) to be scheduled, and implements a basic scheduling logic (e.g., to let each schedulable compute once per second). Another element of generality is given by keeping all contextual data into a single data structure `sensors`, where the basic idea is that any access to context is mediated by a sensor.

More in detail, the behaviour of `DeviceActor` is defined in terms of reactions to a few message types. The acquisition of contextual information is handled through a push-style interface based on two main incoming messages: `SetSensor` for *local sensors* (e.g., position sensors or temperature sensors), and `SetNbrSensor` for *neighbouring sensors* (i.e., those associating data to neighbours). Neighbouring sensors are used to access the current set of neighbours, information relative to neighbours (e.g., the distance to neighbours), and information shared by neighbours (e.g., their gradient value). Upon these, behaviours associated to

```
abstract class DeviceActor[T](c: ActorContext[DeviceProtocol]) extends AbstractBehavior(c) {
  var sensors = Map[String, Any]()

  def senseOrElse[T](name: String, default: => T): T =
    sensors.getOrElse(name, default).asInstanceOf[T]
  def nbrValue[T](name: String): Map[Nbr, T] =
    senseOrElse[Map[Nbr, T]](name, Map.empty).filter(tp => neighbors.contains(tp._1))
  def neighbors: Set[ActorRef[DeviceProtocol]] =
    senseOrElse[Map[Nbr, Long]](Sensors.neighbors, Map(context.self -> currentTime()))
    .filter(tp => currentTime() - tp._2 < RETENTION_TIME).keySet
  def updateNbrTimestamp(nbr: Nbr, t: Long = currentTime()): Unit =
    sensors += Sensors.neighbors -> (nbrValue[Long](Sensors.neighbors) + (nbr -> t))

  def compute(what: String, d: DeviceActor[T]): T // cf. template method's abstract method

  override def onMessage(msg: DeviceProtocol): Behavior[DeviceProtocol] = msg match {
    case SetSensor(sensorName, value) =>
      sensors += (sensorName -> value)
      this
    case SetNbrSensor(name, nbr, value)Behaviors.withTimers { timers =>
     =>
      val sval = sensors.getOrElse(name, Map.empty).asInstanceOf[Map[Nbr, Any]]
      sensors += name -> (sval + (nbr -> value))
      updateNbrTimestamp(nbr)
      this
    case Compute(what) =>
      val result = compute(what, this)
      neighbors.foreach(_ ! SetNbrSensor(what, context.self, result))
      this
    case AddNeighbour(nbr) =>
        context.self ! SetNbrSensor(Sensors.neighbors, nbr, currentTime())
        context.self ! SetNbrSensor(Sensors.nbrRange, nbr, 1.0)
      this
    case RemoveNeighbour(nbr) =>
      context.self ! SetNbrSensor(Sensors.neighbors, nbr, 0)
      this
    case Stop =>
      Behaviors.stopped
    }
  }
}

// then, a DeviceActor computing a gradient can be launched as follows
val a = ctx.spawn(DeviceActor[Double]((ctx,w,d) => {
  val nbrg = d.nbrValue[Double]("gradient")
    .map(n => n._2 + d.nbrSense[Double](Sensors.nbrRange)(n._1)
                    .getOrElse(Double.PositiveInfinity))
    .minOption.getOrElse(Double.PositiveInfinity)
  if(d.senseOrElse("source", false)) 0.0 else nbrg
}), "device-1")
a ! SetSensor("source", true)
a ! AddNeighbour(...)
a ! Compute("gradient")
```

Fig. 2. An improved Akka implementation of a reusable device.

specific control messages like `AddNeighbour` and `RemoveNeighbour` can be easily implemented. Then, the `Compute(what)` message, carrying an indication of `what` has to be computed (to enable multiple computations), is handled by calling the `compute` abstract method, and then communicating the corresponding result to the neighbours by sending a `SetNbrSensor` message. Finally, at the bottom of

Figure 2 it is shown how the gradient computation can be specified, and how an actor computing the gradient can be configured.

4 The ScaFi Akka-based Distributed Middleware

In this section, we present an implementation of a general self-organisation programming system, based on the aggregate computing paradigm [42] and integrated into the *ScaFi toolkit* [17], whose *runtime* (also called a *middleware*) is based on actors, along the lines of the improved design presented in Section 3.2. Interestingly, the design is organised in order to support *distributed* execution of aggregate systems, also according to multiple *architectural styles* (cf. [16,15])—which is important to fully exploit modern infrastructures like the heterogeneous multi-scale computing continua of which the *edge-cloud continuum* is a prominent example [9].

4.1 System Design

A simplified view of the elements participating in an actor-based aggregate computing application is provided by Figure 3.

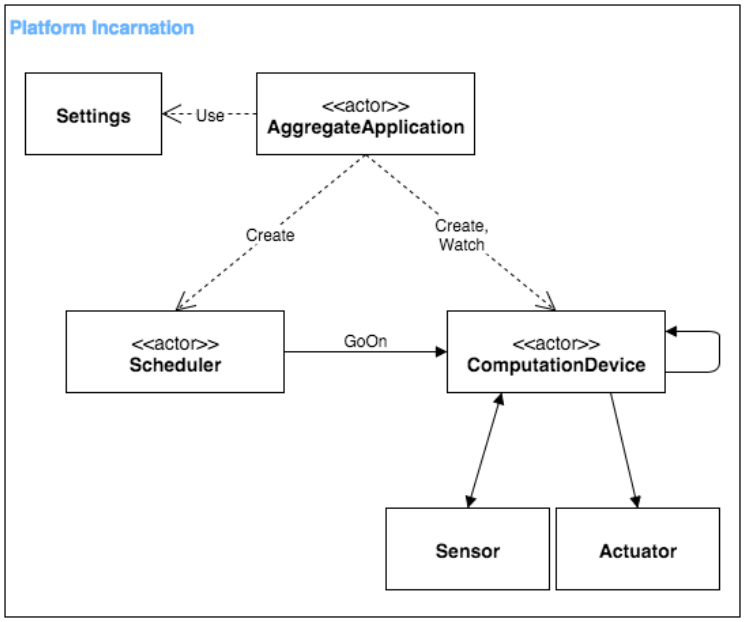

Fig. 3. Structure diagram of the main entities of an aggregate computing system.

Essentially, the key types of elements are:

- `AggregateApplication` – It represents, in any subsystem, a particular aggregate application, as specified by some `Settings`. Also, it works as a supervisor for all the other application-specific actors. This notion is required to properly handle the management of multiple aggregate computations on the same infrastructure.
- `Scheduler` – Optionally, a scheduler may be used to centralise system execution at a system- or subsystem-level.
- `ComputationDevice` – It is a device which is able to carry out some local computation. It communicates with other devices and interacts with `Sensor`s and `Actuator`s (which may be actors as well or not).

Also, note how all these entities are specific to a particular *platform incarnation*, i.e., a concrete set of implementations for the defined types (see also the notion of "incarnation" as an instantiated "family of types" in ScaFi [17]).

Devices Figure 4 shows how devices are modelled. A first key distinction is between actors and actor behaviours. In fact, one design goal is to split a big, articulated behaviour into many small, reusable, composable behaviours. The convention in the diagram is to express message-based interfaces by means of incoming and outcoming messages which are represented as arrows with a filled arrowhead.

By a conceptual point of view, a device must, at minimum, manage its sensors and actuators. Then, in the context of aggregate computing, a device must also interact with its neighbours (`BaseNbrManagementBehavior`); such interaction has not been detailed yet, as it may be somehow different in the peer-to-peer and server-based cases. Also, a computation device executes some program with a certain frequency (here represented by a tick message called `GoOn`, externally or self-sent).

4.2 Server-based Actor Platform

The *server-based platform*, following the client/server architectural style, is depicted in Figure 5. The devices are clients of a central server that owns the information about the *topology* of the aggregate system and is responsible for the propagation of the exports of the devices.

Figure 5 statically describes the message interfaces of device and server:

- Each device registers itself with the server at startup (`Registration`).
- After a computation, a device communicates its newly computed state to the server (`Export`).
- Each device asks the server (`GetNeighbourhoodExports`) for the most recent states of its neighbours (`NeighbourhoodExports`), with some frequency.

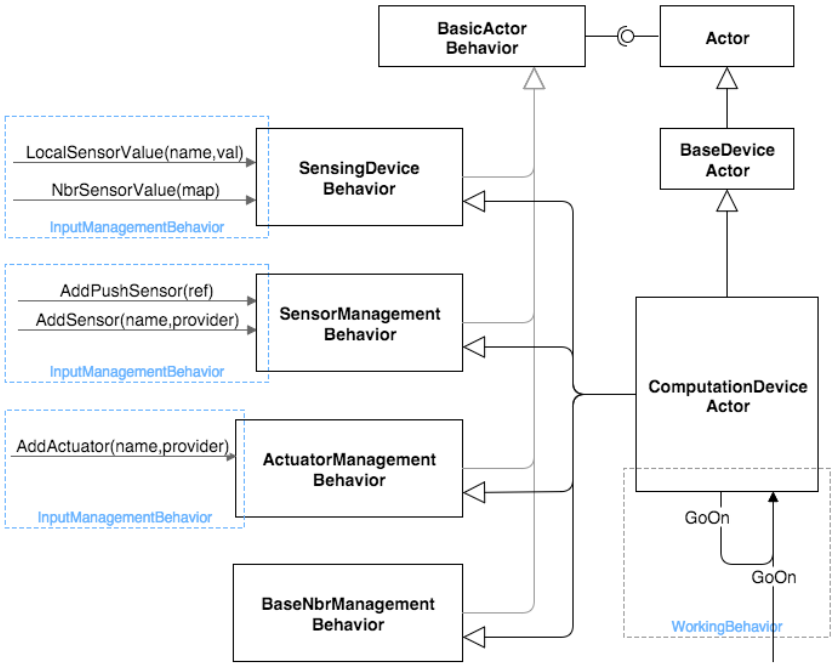

Fig. 4. Structure and interface of device actors.

4.3 Peer-to-peer Actor Platform

The *peer-to-peer platform*, following an ad-hoc architectural style, is shown in Figure 6. Each device, at the end of each computation, propagates its newly computed state (`MsgExport`) directly to all its neighbour actors. Here, the critical point concerns how a device gets acquainted with its neighbours, i.e., by receiving information about a neighbour (`NbrInfo`).

4.4 Actors and Reactive Behaviour

The ScaFi actor platform was implemented using *Akka Classic* framework [35]. In Akka Classic, actors are defined by extending the `akka.actor.Actor` trait and implementing the `receive` method, of type `Receive=PartialFunction[Any,Unit]`, that associates reactions to incoming messages.

An interesting implication of having (reactive) behaviours expressed by `PartialFunctions` is that they *compose*. This composability feature has been extensively used to promote separation of concerns. For example, the device behaviour related to the management of sensors can be kept separated from the behaviour aimed at handling actuators:

```
def SensorManagementBehavior: Receive = {
```

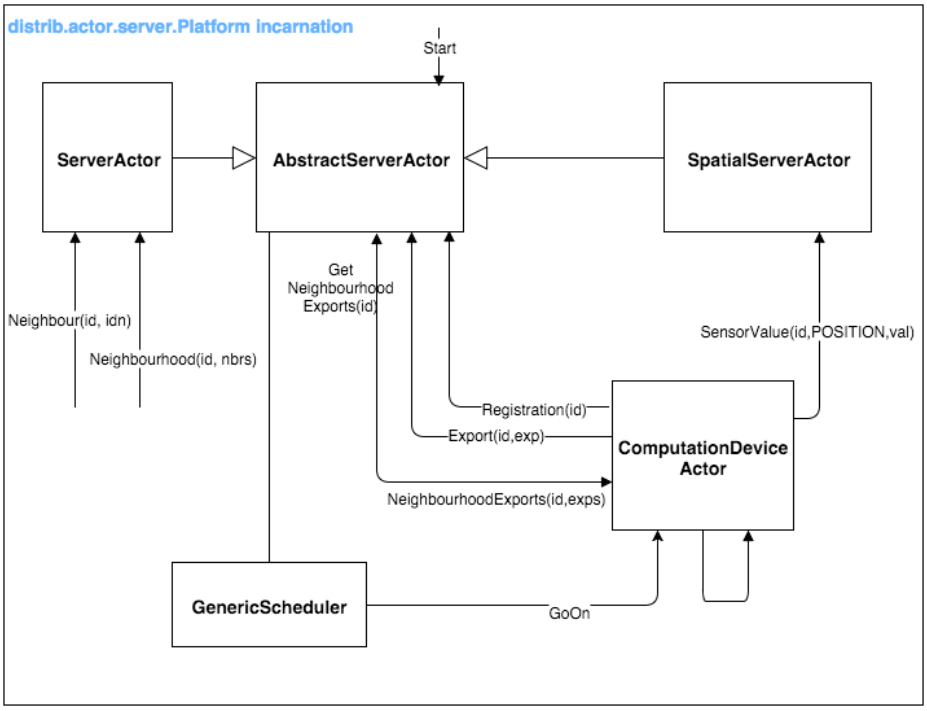

Fig. 5. Key elements and relationships in a server-based actor platform.

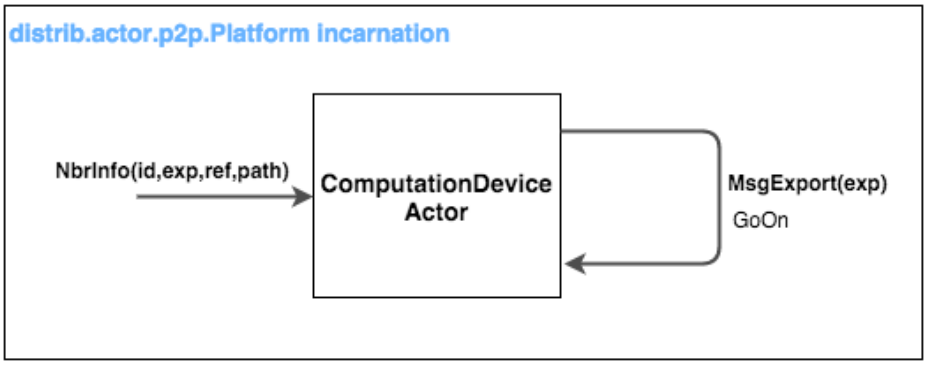

Fig. 6. Key elements and relationships in a peer-to-peer actor platform.

```
  case MsgAddPushSensor(ref) => { ref ! MsgAddObserver(self); ref ! GoOn }
  case MsgAddSensor(name, provider) => setLocalSensor(name, provider)
}

def ActuatorManagementBehavior: Receive = {
  case MsgAddActuator(name, consumer) => setActuator(name, consumer)
}

def CompositeBehavior: Receive =
  SensorManagementBehavior
  .orElse(ActuatorManagementBehavior)
```

Moreover, it is also possible to leverage on *trait stacking* to automatically extend some behaviour by mixing in behaviour traits:

```
trait BasicActorBehavior { selfActor: Actor =>

  def receive: Receive =
    workingBehavior
      .orElse(inputManagementBehavior)
      .orElse(queryManagementBehavior)
      .orElse(commandManagementBehavior)

  def inputManagementBehavior: Receive = Map.empty
  def queryManagementBehavior: Receive = Map.empty
  def commandManagementBehavior: Receive = Map.empty
  def workingBehavior: Receive = Map.empty
}

trait SensorManagementBehavior extends BasicActorBehavior { selfActor: Actor =>
  def SensorManagementBehavior: Receive = { ... }

  override def inputManagementBehavior: Receive =
    super.inputManagementBehavior.orElse(SensorManagementBehavior)

  // ...
}

trait ActuatorManagementBehavior extends BasicActorBehavior { selfActor: Actor =>
  def ActuatorManagementBehavior: Receive = { ... }

  override def inputManagementBehavior: Receive =
    super.inputManagementBehavior.orElse(ActuatorManagementBehavior)

  // ...
}

class DeviceActor extends Actor
  with SensorManagementBehavior
  with ActuatorManagementBehavior { ... }
```

Finally, ScaFi provides an object-oriented façade API for setting up, launching, and managing a running system upon the described actor-based middleware. Please refer to the ScaFi repository[4] and website[5] for further details.

5 Discussion and Future Work

The development of actor-based designs and implementations of self-organising behaviours like the gradient, as well as the experience in research and devel-

[4] https://github.com/scafi/scafi
[5] https://scafi.github.io/

opment of aggregate computing systems, provided some general insights about self-organisation programming. These suggest some general principles (as also indicated by modern software engineering practice) or desiderata for implementations. In particular, we emphasise the following.

Declarativity. The program logic expressing how self-organisation is carried out should be as declarative as possible. This means that the program should abstract from a number of details, e.g. including the following: (i) scheduling of context retrieval and update, (ii) scheduling of computation, (iii) neighbourhood management, (iv) details of message passing (cf. the naive actor design vs. the improved design vs. the ScaFi program), and (iv) application partitioning and deployment (cf. [16]). Aggregate programming in general and ScaFi in particular do support a programming model where such details are abstracted away: this provides great operational flexibility [15].

Composability of behaviour. Another benefit of aggregate programming is compositionality, namely the ability of connecting basic self-organising behaviours (e.g., gradients—cf. Section 2.1) in order to build more complex self-organising behaviours (e.g., channels—cf. Section 2.1). The problem with the actor-based design proposed in Section 3 is that explicitly managing the relationships between computations in terms of message-passing is cumbersome and error-prone[6].

Separation of Concerns. Separating different concerns is a well-known design principle in software engineering, fostering modular design. It is also related to the *Single Responsibility Principle (SRP)*, which suggests that a module (e.g., a class or an actor) should handle a single piece of functionality. As we have seen, it is good to separate certain concerns: e.g., the scheduling concern may be encapsulated into a scheduler (actor)—cf. Section 3.2. However, there is the risk of too much separation, possibly leading to over-complication and inefficiency. In the provided repository, for instance, a "fully destructured" device actor is provided, encapsulating the different concerns (sensor management, neighbourhood management, scheduling management, context management, communication management, and computation) into separate child actors; however, this design turns out to be very complex, due to the need of properly managing the interaction among those inter-related sub-actors.

Propensity to openness and reconfiguration. The kinds of systems we are considering in this chapter, i.e., large homogeneous systems (e.g., swarms, IoT systems, etc.), are generally *open* systems, where devices may easily enter or exit the system (also due to failure, user decisions, and environmental dynamics). Additionally, the execution of such systems may need to be *reconfigured* [18]

[6] A sketch of an actor-based implementation of the channel is given in the provided repository

into different architectural styles (cf. Section 4) in order to optimise for or opportunistically exploit available infrastructure by re-deployment [3,15]. Reconfiguration is typically based on *component models* [32,8,16], but also actors have shown their suitability for dynamically reconfigurable open systems [40]. In [43], the prelude of the *pulverisation model* of aggregate computing systems [16], it was proposed to split the behaviour of a device into sub-actors (handling sensors, actuators, communication, and computation), to be potentially deployable (and relocatable) across different architectures. Different approaches may leverage other kinds of components, e.g., based on microservices or containers [18], hence possibly leveraging actors at the level of their implementation.

Fine-grained execution model. Aggregate computing systems typically work in a round-based fashion, where devices repeatedly execute asynchronous rounds atomically performing *sense–compute–act* steps. Actors, instead, promote the construction of asynchronous reactive dataflow graphs, that may in principle support a finer-grained definition of the execution model where, e.g., the only computations that are re-evaluated are those whose inputs have changed. A first reactive extension to aggregate computing, based on reactive policies and explicit program graphs, has been proposed in [34]. A different approach may exploit the functional reactive programming paradigm [7]. A comparison between these approaches and potential actor-based design may be an interesting future work, to determine more efficient and finely controllable execution strategies, and to possibly also provide general insights about the relationship between self-organisation and reactivity.

Acknowledgements

This work has been partially supported by the MUR PRIN 2020 Project "COMMON-WEARS" (2020HCWWLP) and the EU/MUR FSE REACT-EU PON R&I 2014-2020. This study was carried out within the Agritech National Research Center and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.4 – D.D. 1032 17/06/2022, CN00000022). This publication is also part of the project NODES which has received funding from the MUR – M4C2 1.5 of PNRR with grant agreement no. ECS00000036. This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

References

1. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)
2. Aguzzi, G., Casadei, R., Viroli, M.: Towards reinforcement learning-based aggregate computing. In: ter Beek, M.H., Sirjani, M. (eds.) Coordination Models

and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13271, pp. 72–91. Springer (2022). https://doi.org/10.1007/978-3-031-08143-9_5

3. Arcangeli, J., Boujbel, R., Leriche, S.: Automatic deployment of distributed software systems: Definitions and state of the art. J. Syst. Softw. **103**, 198–218 (2015). https://doi.org/10.1016/j.jss.2015.01.040
4. Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. Comput. Networks **54**(15), 2787–2805 (2010). https://doi.org/10.1016/j.comnet.2010.05.010
5. Audrito, G., Casadei, R., Damiani, F., Viroli, M.: Compositional blocks for optimal self-healing gradients. In: SASO. pp. 91–100. IEEE (2017). https://doi.org/10.1109/SASO.2017.18
6. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Trans. Comput. Log. **20**(1), 5:1–5:55 (2019). https://doi.org/10.1145/3285956
7. Bainomugisha, E., Carreton, A.L., Cutsem, T.V., Mostinckx, S., Meuter, W.D.: A survey on reactive programming. ACM Comput. Surv. **45**(4), 52:1–52:34 (2013). https://doi.org/10.1145/2501654.2501666
8. Ballouli, R.E., Bensalem, S., Bozga, M., Sifakis, J.: Four exercises in programming dynamic reconfigurable systems: Methodology and solution in DR-BIP. In: Leveraging Applications of Formal Methods, Verification and Validation. ISoLA, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11246, pp. 304–320. Springer (2018). https://doi.org/10.1007/978-3-030-03424-5_20
9. Bittencourt, L.F., Immich, R., Sakellariou, R., da Fonseca, N.L.S., Madeira, E.R.M., Curado, M., Villas, L., DaSilva, L.A., Lee, C., Rana, O.F.: The internet of things, fog and cloud continuum: Integration and challenges. Internet Things **3-4**, 134–155 (2018). https://doi.org/10.1016/j.iot.2018.09.005
10. Boissier, O., Bordini, R.H., Hubner, J., Ricci, A.: Multi-agent oriented programming: programming multi-agent systems using JaCaMo. Mit Press (2020)
11. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence: From Natural to Artificial Systems. Santa Fe Institute Studies in the Sciences of Complexity, Oxford University Press, Inc. (1999)
12. Brambilla, M., Ferrante, E., Birattari, M., Dorigo, M.: Swarm robotics: a review from the swarm engineering perspective. Swarm Intell. **7**(1), 1–41 (2013). https://doi.org/10.1007/s11721-012-0075-2
13. Busoniu, L., Babuska, R., Schutter, B.D.: A comprehensive survey of multiagent reinforcement learning. IEEE Trans. Syst. Man Cybern. Part C **38**(2), 156–172 (2008). https://doi.org/10.1109/TSMCC.2007.913919
14. Casadei, R.: Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. ACM Computing Surveys (Jan 2023). https://doi.org/10.1145/3579353
15. Casadei, R., Fortino, G., Pianini, D., Placuzzi, A., Savaglio, C., Viroli, M.: A methodology and simulation-based toolchain for estimating deployment performance of smart collective services at the edge. IEEE Internet Things J. **9**(20), 20136–20148 (2022). https://doi.org/10.1109/JIOT.2022.3172470
16. Casadei, R., Pianini, D., Placuzzi, A., Viroli, M., Weyns, D.: Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment. Future Internet **12**(11), 203 (Nov 2020). https://doi.org/10.3390/fi12110203

17. Casadei, R., Viroli, M., Aguzzi, G., Pianini, D.: Scafi: A scala DSL and toolkit for aggregate programming. SoftwareX **20**, 101248 (2022). https://doi.org/10.1016/j.softx.2022.101248
18. Coullon, H., Henrio, L., Loulergue, F., Robillard, S.: Component-based distributed software reconfiguration: A verification-oriented survey. ACM Comput. Surv. (may 2023). https://doi.org/10.1145/3595376, just Accepted
19. De Nicola, R., Jähnichen, S., Wirsing, M.: Rigorous engineering of collective adaptive systems: special section. Int. J. Softw. Tools Technol. Transf. **22**(4), 389–397 (2020). https://doi.org/10.1007/s10009-020-00565-0
20. De Wolf, T., Holvoet, T.: Emergence versus self-organisation: Different concepts but promising when combined. In: Brueckner, S., Serugendo, G.D.M., Karageorgos, A., Nagpal, R. (eds.) Engineering Self-Organising Systems, Methodologies and Applications (ESOA workshop, AAMAS conference). Lecture Notes in Computer Science, vol. 3464, pp. 1–15. Springer (2004). https://doi.org/10.1007/11494676_1
21. Franklin, S., Graesser, A.C.: Is it an agent, or just a program?: A taxonomy for autonomous agents. In: Müller, J.P., Wooldridge, M.J., Jennings, N.R. (eds.) Intelligent Agents III, Agent Theories, Architectures, and Languages, ECAI '96 Workshop (ATAL), Budapest, Hungary, August 12-13, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1193, pp. 21–35. Springer (1996). https://doi.org/10.1007/BFb0013570
22. Gamma, E., Helm, R., Johnson, R., Johnson, R.E., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH (1995)
23. Gershenson, C.: Design and control of self-organizing systems. CopIt Arxives (2007)
24. Hewitt, C.: A universal modular actor formalism for artificial intelligence. Proc. of IJCAI, 1973 (1973)
25. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). https://doi.org/10.1109/MC.2003.1160055
26. Koster, J.D., Cutsem, T.V., Meuter, W.D.: 43 years of actors: a taxonomy of actor models and their key properties. In: Clebsch, S., Desell, T., Haller, P., Ricci, A. (eds.) Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016. pp. 31–40. ACM (2016). https://doi.org/10.1145/3001886.3001890
27. Mamei, M., Zambonelli, F., Leonardi, L.: Co-fields: A physically inspired approach to motion coordination. IEEE Pervasive Computing **3**(2), 52–61 (2004). https://doi.org/10.1109/MPRV.2004.1316820
28. Martius, G., Herrmann, J.M.: Variants of guided self-organization for robot control. Theory Biosci. **131**(3), 129–137 (2012). https://doi.org/10.1007/s12064-011-0141-0
29. Montesi, F.: Choreographic programming. Ph.D. thesis (2014), https://pure.itu.dk/ws/files/78733848/m13_phd.pdf
30. Nagpal, R., Shrobe, H.E., Bachrach, J.: Organizing a global coordinate system from local information on an ad hoc sensor network. In: IPSN. Lecture Notes in Computer Science, vol. 2634, pp. 333–348. Springer (2003). https://doi.org/10.1007/3-540-36978-3_22
31. Nicola, R.D., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. ACM Trans. Auton. Adapt. Syst. **9**(2), 7:1–7:29 (2014). https://doi.org/10.1145/2619998

32. Nicola, R.D., Maggi, A., Sifakis, J.: The DReAM framework for dynamic reconfigurable architecture modelling: theory and applications. Int. J. Softw. Tools Technol. Transf. **22**(4), 437–455 (2020). https://doi.org/10.1007/s10009-020-00555-2
33. Parunak, H.V.D., Brueckner, S.A.: Software engineering for self-organizing systems. Knowl. Eng. Rev. **30**(4), 419–434 (2015). https://doi.org/10.1017/S0269888915000089
34. Pianini, D., Casadei, R., Viroli, M., Mariani, S., Zambonelli, F.: Time-fluid field-based coordination through programmable distributed schedulers. Logical Methods in Computer Science **Volume 17, Issue 4** (Nov 2021). https://doi.org/10.46298/lmcs-17(4:13)2021
35. Roestenburg, R., Williams, R., Bakker, R.: Akka in action. Simon and Schuster (2016)
36. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2), 14:1–14:42 (2009). https://doi.org/10.1145/1516533.1516538
37. Samarasinghe, D., Lakshika, E., Barlow, M., Kasmarik, K.: Automatic synthesis of swarm behavioural rules from their atomic components. In: Aguirre, H.E., Takadama, K. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2018, Kyoto, Japan, July 15-19, 2018. pp. 133–140. ACM (2018). https://doi.org/10.1145/3205455.3205546
38. Singh, V.K., Singh, G., Pande, S.: Emergence, self-organization and collective intelligence - modeling the dynamics of complex collectives in social and organizational settings. In: UKSim. pp. 182–189. IEEE (2013). https://doi.org/10.1109/UKSim.2013.77
39. Trianni, V.: Evolutionary Swarm Robotics - Evolving Self-Organising Behaviours in Groups of Autonomous Robots, Studies in Computational Intelligence, vol. 108. Springer (2008). https://doi.org/10.1007/978-3-540-77612-3
40. Varela, C.A., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN Notices **36**(12), 20–34 (2001). https://doi.org/10.1145/583960.583964
41. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul. **28**(2) (2018). https://doi.org/10.1145/3177774
42. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. J. Log. Algebraic Methods Program. **109** (2019). https://doi.org/10.1016/j.jlamp.2019.100486
43. Viroli, M., Casadei, R., Pianini, D.: On execution platforms for large-scale aggregate computing. In: Lukowicz, P., Krüger, A., Bulling, A., Lim, Y., Patel, S.N. (eds.) Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp Adjunct 2016, Heidelberg, Germany, September 12-16, 2016. pp. 1321–1326. ACM (2016). https://doi.org/10.1145/2968219.2979129
44. Wolf, T.D., Holvoet, T.: Designing self-organising emergent systems based on information flows and feedback-loops. In: SASO. pp. 295–298. IEEE Computer Society (2007). https://doi.org/10.1109/SASO.2007.16
45. Wooldridge, M.J.: An Introduction to MultiAgent Systems, Second Edition. Wiley (2009)
46. Zhang, K., Yang, Z., Basar, T.: Multi-agent reinforcement learning: A selective overview of theories and algorithms. CoRR **abs/1911.10635** (2019), http://arxiv.org/abs/1911.10635