Engineering collective intelligence at the edge with aggregate processes

(Article begins on next page)

25 April 2024

Engineering Collective Intelligence at the Edge
with Aggregate Processes

Roberto Casadei[a], Mirko Viroli[a], Giorgio Audrito[b],
Danilo Pianini[a], Ferruccio Damiani[b]

[a]*Alma Mater Studiorum—Università di Bologna, Italy*
{roby.casadei,mirko.viroli,danilo.pianini}@unibo.it

[b]*Università di Torino, Italy*
{giorgio.audrito,ferruccio.damiani}@unito.it

Abstract

Edge computing promotes the execution of complex computational processes without the cloud, i.e., on top of the heterogeneous, articulated, and possibly mobile systems composed of IoT and edge devices. Such a pervasive smart fabric augments our environment with computing and networking capabilities. This leads to a complex and dynamic ecosystem of devices that should not only exhibit individual intelligence but also *collective intelligence*—the ability to take group decisions or process knowledge among autonomous units of a distributed environment. Self-adaptation and self-organisation mechanisms are also typically required to ensure continuous and inherent toleration of changes of various kinds, to distribution of devices, energy available, computational load, as well as faults. To achieve this behaviour in a massively distributed setting like edge computing demands, we seek for identifying proper abstractions, and engineering tools therefore, to smoothly capture collective behaviour, adaptivity, and dynamic injection and execution of concurrent distributed activities. Accordingly, we elaborate on a notion of "aggregate process" as a concurrent collective computation whose execution and interactions are sustained by a dynamic team of devices, whose spatial region can opportunistically vary over time. We ground this notion by extending the aggregate computing model and toolchain with new constructs to instantiate aggregate processes and regulate key aspects of their lifecycle. By virtue of an open-source implementation in the ScaFi framework, we show basic programming examples as well as case studies of edge computing, evaluated by simulation in realistic settings.

Keywords: computational collective intelligence, self-organisation, distributed computing, aggregate programming, computational fields, multi-agent systems

1. Introduction

Emerging scenarios like pervasive computing, Internet of Things (IoT), and cyber-physical systems (CPS) are leading towards a new reference computa-

tional *fabric* made of dense, large-scale networks of heterogeneous devices. In such contexts, software services are naturally highly contextual, and hence fundamentally related to their space-time situation and physical environment [1]. They opportunistically leverage the available parts of such fabric, dynamically exploiting its sensing, actuation, storage, computation, and networking capabilities [2, 3].

In this paper, we seek to unveil the potential of such digitally empowered ecosystems in supporting *(computational) collective intelligence*, i.e., the form of artificial intelligence [4] dealing with group decision making or knowledge processing in distributed environments of autonomous situated entities [5]. Most specifically, we follow ideas of *swarm intelligence* [6], relying on the repeated interaction of simple information-processing units. We shall design proper abstractions and development techniques to smoothly express *collective* behaviour leading to intelligent activities that can be transparently executed on opportunistic formations of devices [7]. In doing so, we then target so-called *edge computing*: a paradigm where computation is performed at the "network edge", close to data sources and users [8].

Openness and dynamism require distributed collective activities to be dependable, self-adaptive and self-organising in order to maintain coherence and functionality across unpredictable and inevitable context changes and adversary events. They should also opportunistically activate wherever and whenever their existence conditions hold—whether they are by-design or emergent. For instance, for collaborative smartphone-based applications in a smart city, such activities may include: a gossip process by which people in a plaza share comments, a guidance process to make a group of friends gather in a convenient point, a dispersal process for people creating bloat, a process to advertise one's presence to nearby users for the next minute, a process providing crowd-aware directions towards a point of interest, and so on [9, 10, 11, 8]. Similarly, a swarm of robots (Figure 1) to be engaged in an exploration mission could be programmed to autonomously perform multiple concurrent activities that are collective in nature. Such concurrent tasks may include: coordinated movement in flock formation [12]; estimation of a physical quantity of interest [13, 14], such as the humidity level or risk of fire (cf. case study in Section 6.2); collective decision-making [15], to determine a team of robots for handling certain sub-tasks, based on available energy and capabilities of the robots; and so on.

According to this vision, we present the concept of *aggregate process*, denoting a distributed computation sustained by a dynamic aggregation of devices. By this abstraction we shall target the design of transient collective activities, which may concurrently span and overlap over the fabric created by large-scale deployment of possibly mobile devices. Aggregate processes are intended to capture dynamicity and context-orientation in an intrinsically resilient fashion, taking an *aggregate* stance—with "aggregate" having the meaning of "pertaining to a collective" [9, 16]. Similar to multi-agent organisational approaches [17], this notion fosters a vision of smart distributed environments like a *society* or *ecosystem*, where services and processes are seen as cooperative activities that involve and influence groups of agents.

2

To formally capture the features of aggregate processes, and experiment with mechanisms to handle their lifecycle (process creation, disposal, logic and interaction), we adopt as basis framework the *aggregate computing* paradigm [9, 16]. This is a functional approach to collective adaptive system programming based on the notion of *computational field* [18], namely, a time-evolving distributed structure mapping devices to computational values. The essence of aggregate computing is currently captured by the *field calculus* [18], whereby the self-organising, collective behaviour of an ensemble is declaratively and compositionally expressed as pure functions from fields to fields. Aggregate processes are supported in the field calculus by the primitive construct `spawn`, yielding a field that, across space and time, combines several independent but interacting "computational bubbles" (process instances). Programming constructs to work with aggregate processes are implemented in SCAFI[1] [19, 20], a Scala-based aggregate programming toolkit. The proposed implementation is used to showcase the expressiveness of the notion and to empirically evaluate the proposed abstraction through simulation of three paradigmatic case studies: (i) coordination of communications in a mobile, ad hoc-network; (ii) consensus reaching on a drone swarm; and (iii) self-discovery of nodes provisioning edge-replicated services.

In summary, this work provides the following major contributions:

1. presentation of the "aggregate process" abstraction to capture a dynamic, concurrent activity carried out by a dynamic ensemble of devices;
2. implementation in the SCAFI aggregate programming framework, through the primitive construct `spawn`;
3. design of an Application Program Interface (API) and idioms for programming with aggregate processes in SCAFI;
4. evaluation of the expressiveness and benefits of aggregate processes through simulated case studies based of three scenarios in pervasive computing, swarm intelligence, and edge computing.

With respect to the conference paper [21], where construct `spawn` is formalised, this work elaborates on the engineering perspective, by: providing more details, explanations, and much deeper comparison with respect to related work; discussing motivation, requirements, and characteristics of the aggregate process abstraction; covering implementation details in SCAFI; providing a full account of programming techniques for building applications with aggregate processes; and showing practical expressiveness through a whole new case study of edge service discovery.

The remainder of this paper is organised as follows. Section 2 gives an informal introduction to the aggregate process concept, by providing motivation, desired characteristics, and prominent features. Section 3 provides background for the rest of the paper, by introducing the aggregate computing paradigm and the SCAFI language. Section 4 explains how aggregate computing and SCAFI
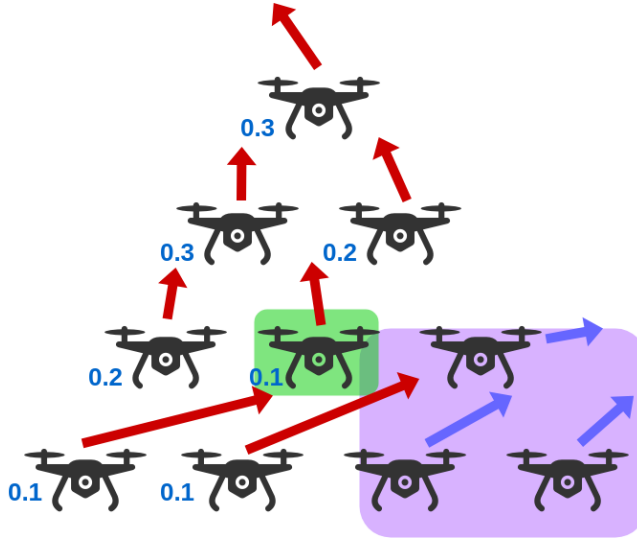
---

[1]`http://scafi.github.io/`

Figure 1: A drone swarm running concurrent collective computations. The swarm moves in flocking formation in the direction set by the "driver" drone, while estimating some physical quantity. At the same time, a continuous leader election process is in place (cf., green drone). At some point, a team (violet bubble) leaves the swarm for a sub-task, and the flock re-adapts to the change in the formation.

are extended to support the aggregate process abstraction. Section 5 covers programming with aggregate processes, explaining the semantics of the abstraction and showing how process-based APIs can be developed. Section 6 provides evaluation of aggregate processes through synthetic experiments: here, we show that processes provide practical advantages both in terms of performance and in terms of programmability. Section 7 includes a detailed coverage of related work. Finally, Section 8 concludes the paper.

2. Aggregate Processes: Concepts

We start focussing on the notion of *aggregate processes*, as key abstraction for modelling:

> dynamic, context-driven and collective activities that concurrently span and overlap into a possibly mobile, large-scale collection of situated, computational devices—which we call an *aggregate* or *ensemble*.

Figure 2 describes the role of aggregate processes in IoT systems, in terms of relationships with typical entities involved, which are the situated devices (*things*) and new first-class citizens like aggregates (collectives of things).
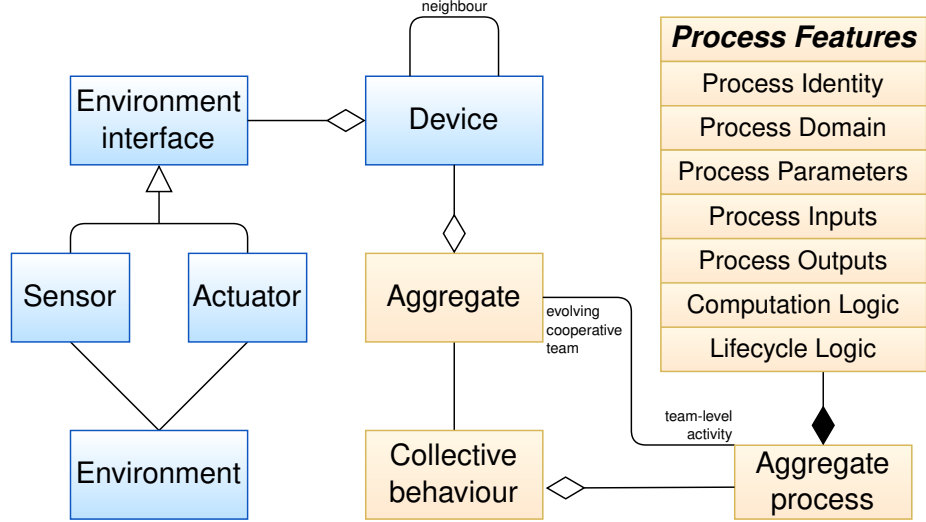
4

Figure 2: Logical UML model of IoT systems, comprising first-class aggregates cooperatively playing some aggregate behaviour which may include multiple concurrent aggregate processes. Colours are used to discriminate between individual (blue) and collective (orange) concepts.

2.1. Requirements

In order to be more systematic in the characterisation of our aggregate process abstraction, we make a set of requirements and desiderata explicit. These unify the aforementioned vision with pragmatic aspects of the software engineering practice:

- *Collective stance* — To promote pervasive adaptation, aggregate processes must abstract the individual device and seamlessly regulate the behaviour of an ensemble across scales, density, and heterogeneity.

- *Dynamicity and context-orientation* — Aggregate processes should conveniently support the implementation of dynamically evolving, distributed, and possibly spatio-temporally situated activities where context plays a major role and continuous change is the norm.

- *Intrinsic resiliency* — Aggregate processes must react to change and failure; in particular, implementations should provide formal guarantees about independence to large classes of environmental dynamics and faults.

- *Opportunistic resource exploitation* — Aggregate processes should support dynamic execution strategies across elastic and heterogeneous infrastructure.

- *Conceptual, methodological, and technological integration* — Aggregate processes should integrate with mainstream paradigms, development techniques and tools.

Multiple aspects and perspectives – structural, behavioural, and interactional – need to be considered to fully characterise an aggregate process. In this paper, we shall abuse for convenience the term *process* to refer to both process *types* and process *instances*, i.e., concrete, living occurrences of process types. For example, a process type can model a gossip activity, its instances being actual executions—this is similar to the class versus object distinction in object-oriented programming.

When specifying an aggregate process, a designer should be generally able to control the following aspects:

- *Process lifecycle* — It includes process *generation* (where and when a process is to be spawned) and *destruction* (shutdown logic and teardown dynamics), which are generally based on recognition of contextual events.

- *Process identity* — Since there could be multiple process instances of a given process type, it is important to define a way to identify them in order to correlate process-scoped activities. This also supports understanding whether different computations belong to the same process and can hence be *merged.*

- *Process structure* — Mechanisms are needed to regulate the *spatio-temporal extension of a process* or, similarly, define the dynamic *domain* of devices participating in the process. This is related to defining a *shape* and *boundary* for the process.

- *Process behaviour* — It covers mechanisms to define the core *logic* of a process, representing an *input-output transformation* in terms of a set of *parameters.*

- *Inter-process interaction* — It covers the ways by which different collective computations can affect each other, e.g., by "piping" the input of one to the output of another, or by indirectly causing the generation of a process by issuing a particular event.

Metaphorically, aggregate processes can be imagined as sorts of "computational bubbles" (Figure 3), sustained by collectives of devices, that spring out, stretch, perform some work, and vanish across space and time.

3. Background: Aggregate Computing and ScaFi

In this section, we describe the aggregate computing approach and SCAFI implementation, which can provide a ground to meet the requirements stated in Section 2.1, and which therefore represent a natural basis upon which developing our aggregate process notion.
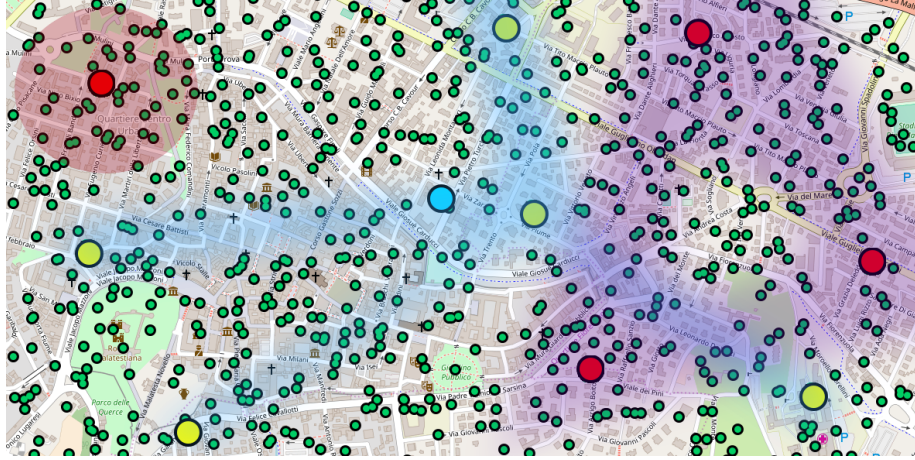
Figure 3: Aggregate processes are like dynamically evolving, distributed computational "bubbles", adapting as devices move or enter/leave the system. So, for instance: the (top, left) red bubble could carry out a process collecting information from an area into a sink device, vanishing upon reception of the final value; the "x"-shape, blue process could consist of two point-to-point communication channels supporting P2P interaction when there is no global connectivity (cf., Section 6.1), closing upon hang-up; finally, the purple triangle could represent a monitoring process keeping statistics about the people or vehicles crossing a particular area of a city.

3.1. Aggregate Computing

Aggregate Computing [9, 16] is an engineering paradigm (see Figure 4) for collective adaptive systems. In a nutshell, it provides a functional programming model that enables three main elements:

1. abstraction and composition of collective behaviours;
2. expression of the self-adaptive, self-organising logic of an ensemble;
3. flexibility of mapping aggregate computation onto the available infrastructure [22].

A key merit of the approach is that it turns the self-organisation and collective intelligence problem into a programming language and middleware problem.

3.1.1. Foundations: Logical Model and Field Calculus

The programming model is formally grounded in the *field calculus* [18], a core language capturing the essence of "continuous distributed computation" through *(computational) fields* and the very principles for "engineering emergent behaviour" [18]. Fields are time-evolving maps from a domain of devices to computational values, so they are collective and dynamic in nature.

In this framework, one defines a single *aggregate program* that is conceptually run on an entire *aggregate*. An aggregate is a logical network of asynchronously computing devices connected by some *neighbouring relationship* and collaboratively playing the same aggregate program. A simple aggregate is shown as a network in Figure 5.
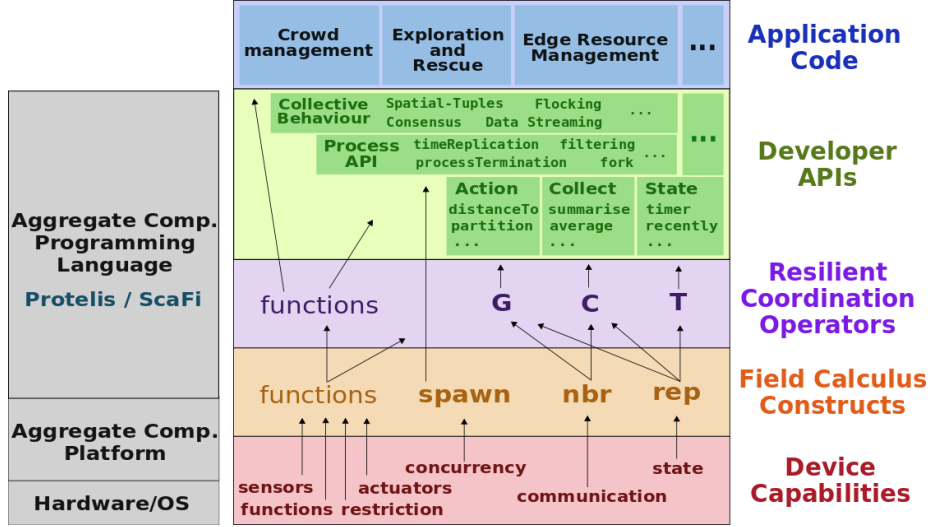
Figure 4: Aggregate Computing Engineering Stack—the aggregate process concept, captured by the `spawn` construct, is the framework extension discussed in this paper that opens to the dimension of concurrency—adapted from [9].
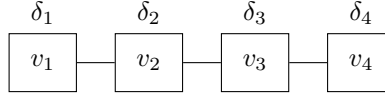
Figure 5: Simple network representing a logical aggregate system. We denote devices through square nodes. Value v_i denotes the most recent value computed by device δ_i. The neighbouring relationship is assumed to be symmetric and denoted through solid lines connecting nodes.

As we will see in Section 3.2, an aggregate program is a standard computation plus calls to field operations that basically express a way to compositionally (i) handle state, (ii) interact with neighbours, and (iii) branch computation, i.e., dynamically entering a scope. An aggregate program specifies collective behaviour in terms of both computation and neighbour-to-neighbour interaction.

3.1.2. Platform Assumptions

A possibly distributed platform is assumed to be available that provides each device of the aggregate with local essential computational/interaction mechanisms, namely:

1. *Neighbourhood* — In each moment of time, each device has a neighbourhood defined as the set of devices it can send messages to. Neighbourhood can be static or change over time to reflect, e.g., mobility and faults.

2. *Sensors* — Upon request, a device can obtain values from local sensors, which model access to the observable part of the environment. Sensors are typically assumed to be the same across all devices. For instance, in

the following we shall use sensors for getting the local `id` of a device, or the perceived temperature.

3. *Actuators* — Upon request, a device can perform an action directed towards itself or the environment.

4. *Message reception* — During operation, devices exchange messages. In each moment of time, any device can retrieve a map from neighbours to the most recent messages received from them; i.e., older messages are discarded.

5. *Message broadcast* — Upon request, any device can broadcast a message to all its neighbours. Message exchange is assumed to be asynchronous and order-preserving. Loss of messages is handled by (possibly transient) changes in neighbours.

6. *State* — Each device has a local memory to make data persist over time. This is typically structured in a dynamic set of typed variables.

7. *Computation* — Each device breaks computations in small pieces called *computational rounds*. Rounds are assumed to be terminating and short, relying on locally available services as described above, and producing a possibly structured value as result. Such a value could be used to feed actuators, for instance.

8. *Scheduling* — Computational rounds are triggered by a local scheduler. Overall, scheduling is generally assumed to be asynchronous and fair, with devices possibly firing at different speed. Also, it is assumed that a round is scheduled only when the previous one is over.

We also note that an aggregate is, first of all, a *logical* network of devices that can be mapped variously to the available *physical* network of computing nodes [22]. In simple deployments, there could be a 1-to-1 mapping from logical to physical devices: e.g., in a robot swarm, each robot could individually execute the aggregate program against its local context.

We summarise the aggregate computing meta-model in Figure 6.

3.1.3. Execution Model

The missing piece to enable emergence of collective intelligence is an execution model in which each device d_i computes the overall aggregate program by organising computational rounds as follows:

1. perceives the local context c_i, given by a sample of the status of the environment, obtained by sensors, and the messages from neighbours;

2. re-evaluates the aggregate program against the local context, obtaining an output v_i and a "coordination message" e_i, called an *export*;

3. acts upon the environment through actuators, as described by the program;

4. sends the export to neighbours—which is basically a way to inform the surroundings about a change in the local context, effectively supporting local-to-global state evolution.
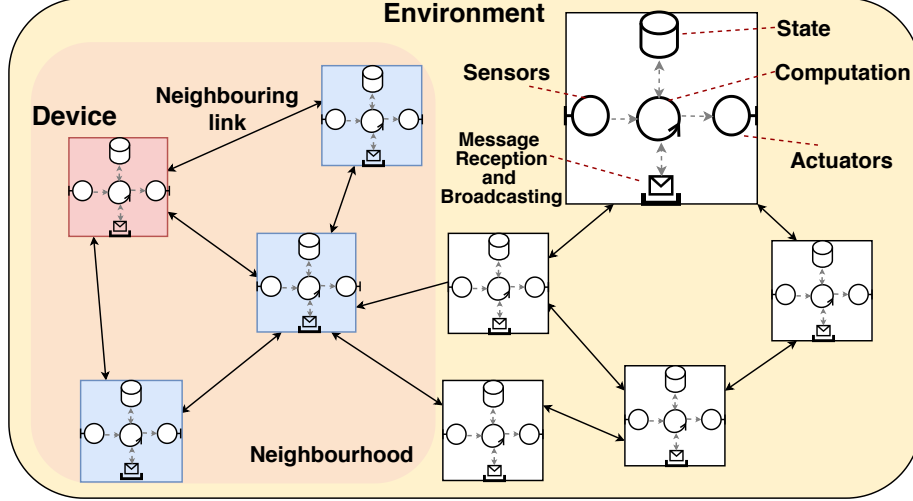
9

Figure 6: A pictorial representation of the aggregate computing meta-model. An aggregate system is a set of devices connected through a neighbouring relationship and situated in some environment. A device has state, sensors, actuators, a computational part, and means to send and receive messages to/from neighbours. Dynamically, devices follow the protocol discussed in Section 3.1.3, collectively behaving as described by the aggregate program they compute.

The very details of the execution model, both within a single device and among devices, are application-specific. Notice that, since rounds of different devices are usually asynchronous, aggregate programs tend to be used to express how an entire aggregate progressively adapts to changes of the environment and the system itself (cf., failure, mobility) and produces eventually consistent responses.

An example of system evolution is given by an *event structure*—see Figure 7. An event structure is a graph of events (corresponding to computational rounds of particular devices), with a partially-ordering causality relation. We implicitly use "columns" of the x-axis (which we call *space*) to label events with device identifiers; so, for instance, the event labelled with δ_1 and all the events below it in the same column denote events happening in the device δ_1. Edges are given "export labels", i.e., symbols denoting particular exports. Notice that, given an event, all the outgoing edges are given the same export label, which represents the export produced by evaluating the aggregate program in the corresponding round. Also, notice that the same export is sent to all the neighbours, including the device itself. Edge labels can be straightforwardly inferred from an event structure: e_{dr} is the export yield at the r-th round performed by device d. In the following, we will omit export labels to improve readability and rather focus on output values written inside the circles.

Devices may execute rounds at different frequencies, and no synchronisation is needed. Still, the timings between rounds at different devices and communication delays may affect the dynamics of distributed algorithms. However, there exist results regarding convergence of field computations (cf., self-
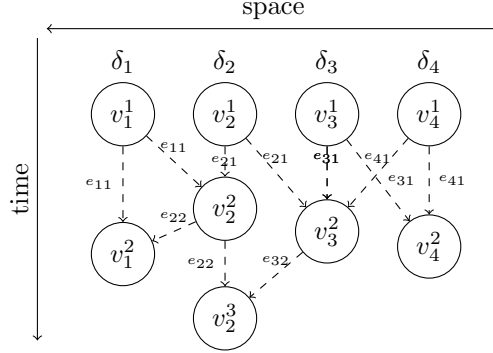
Figure 7: Example of an event structure. Each circle node represents a round execution event, v_i^j denotes the output of device δ_i at its j-th round, and dashed arrows denote an event has observed the export produced at the source event (which captures both state preservation between subsequent rounds of the same device as well as communication between different neighbour devices).

stabilisation [23]). For a deep coverage of Aggregate Computing and the field calculus, refer to [16, 18]. In the rest of the background, the aggregate paradigm is described through its SCAFI implementation.

3.2. SCAFI—*Aggregate Programming in Scala*

SCAFI[2] (SCAla Computational FIelds) is a development toolkit for aggregate systems in the Scala programming language. It provides a Scala-internal domain-specific language (DSL) – i.e., an API masked as an "embedded language" – and library of functions for programming with fields, as well as other development tools (e.g., for simulation).

SCAFI, as an embedded DSL, inherits the syntax and semantics of its host language. Therefore, for the sake of self-containment, we briefly point out in Appendix A some relevant syntactic and semantic aspects of Scala that are used in the SCAFI examples throughout the paper. Notice that while certain features improve the aesthetic quality of the DSL, others may facilitate or enable implementation of particular mechanisms and checks.

In SCAFI, the field constructs are captured by the `Constructs` interface, shown in Figure 8. There, functions (except `mid`) are generic in a type parameter `A`, syntax `=>T` denotes a by-name parameter and syntax `T=>R` denotes a functional type. In a nutshell:

- `rep` captures state evolution, starting from an `init` value that is updated each round through `fun`;

- `nbr` captures communication, of its `expr` value, with neighbours; it is used only inside the argument `expr` of `foldhood`, which supports data

11

```
trait Constructs {
  def rep[A](init: => A)(fun: A => A): A
  def nbr[A](expr: => A): A
  def foldhood[A](init: => A)(acc: (A, A) => A)(expr: => A): A
  def branch[A](cond: => Boolean)(th: => A)(el: => A)

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

Figure 8: Interface modelling field constructs in SCAFI.

```
class MyProgram extends AggregateProgram with BlockG {
  type MainResult = Double // type of result
  def main: MainResult = {
    ???
  }
}
```

Figure 9: An example of aggregate program in SCAFI.

aggregation of neighbourhood-dependent data to single values, through the input accumulator function `acc`;

- `branch` captures domain partitioning, or space-time branching;

- `mid` is a built-in sensor providing the identifier of devices;

- `sense` abstracts access to local sensors; and

- `nbrvar` abstracts access to neighbouring sensors that behave similarly to `nbr` but are provided by the platform.

This interface is implemented by abstract class `AggregateProgram`, which in turn provides its subclasses with access to the field constructs. So, in SCAFI, an aggregate program is merely a class that implements `AggregateProgram` and defines a `main` expression. An example is provided in Figure 9. Notice that library modules such as `BlockG` can be imported through trait inheritance via the `with` keyword.

In order to obtain an actual distributed aggregate system, the aggregate computing metamodel and execution model presented in Section 3.1 are to be implemented by proper platform software, such as SCAFI's Akka actor-based platform [20]. By having any logical device execute the aggregate protocol with a given `AggregateProgram`, which is the same for every device, one effectively gets an operational aggregate system.
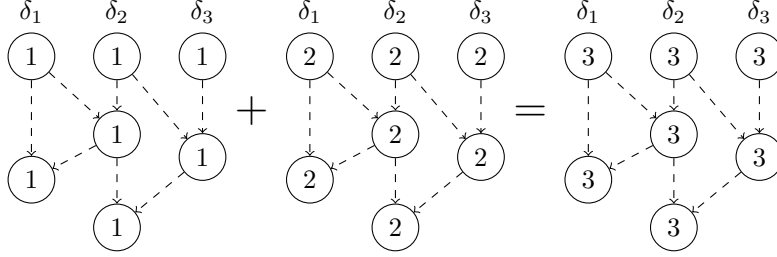
Figure 10: Addition: example of the global interpretation of a field expression.

3.3. Examples

In this section, we explain the field constructs in detail, by examples. To do so, we introduce all the elements needed to implement a simple but illustrative and important example of edge computing. Following the approach in [9], this is a collective behaviour that builds a distributed data structure – i.e., a gradient field – for navigating agents in space through paths of minimal lengths that avoid certain obstacle areas. This example also showcases our approach to edge intelligence. Indeed, it enacts a form of collective intelligence whereby the entire collective globally adapts to target points, dis/appearance of obstacles, and mobility of nodes (including disappearance of nodes by failure or appearance by joining, as in open systems). For such an example, we need mechanisms to: work with field values, including sensor values; handle collective state, through `rep`; observe neighbourhoods, through `foldhood` and `nbr`; distinguish roles and behaviours, through conditional constructs `mux` and `branch`; and combine these to process and propagate information in a network.

In SCAFI a usual expression such as, for instance,

```
1 + 2
```

is to be seen as a *constant* (i.e., not changing over time) and *uniform* (i.e., not changing across space[3]) field holding local value `3` at any point of the space-time domain. More specifically, as shown in Figure 10, this denotes a global expression where a field of `1`s and a field of `2`s are summed together through the field operator `+`, which works like a point-wise application of its local counterpart.

A constant field does not need to be uniform. For instance, given a static network of devices, then

```
mid()
```

denotes the field of device identifiers, exemplified in Figure 11, which does not change across time but does vary in space. We assume devices do not change

---

[3]For *space*, we generally mean the *logical* space given by the graph where nodes are devices and edges represent channels of communication reified by an application-specific neighbouring relationship. For situated systems, however, this representation naturally maps to *physical* space.
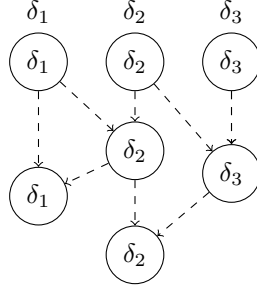
Figure 11: Field of device identifiers, obtained through `mid`.
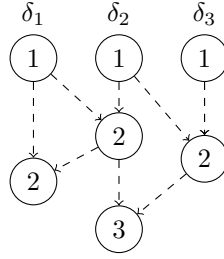
Figure 12: Example field of round counting. At any event of a given device, the output is the value of the previous event at the same device incremented by one.

their identifier for their whole lifetime and that identifiers are unique. On the other hand, expression

```
sense[Double]("temperature")
```

is used to represent a field of temperatures, obtained by collectively querying the local temperature sensors over space and time, which is in general non-constant and non-uniform.

Fields changing over time can also be programmatically defined by the `rep` operator; for instance, expression

```
// Initially 0; state is incremented at each round
rep(0){ x => x + 1 } // Equally expressed in Scala as: rep(0)(_ + 1)
```

counts how many rounds each device has executed—see Figure 12. Indeed, the first time the `rep` expression is evaluated, `x` is bound to `0`, and the expression evaluates to `0+1=1`; the next round on the same device, `x` is bound to `1` (i.e., to the value of the expression in the previous round) and the expression evaluates to `1+1=2`; and so on. It is still a non-uniform field since the update phase and frequency of the devices may vary both between devices and across time for a given device.

Collective intelligence can be programmed by letting the local affect the global and vice versa. This operation is supported through `foldhood` and `nbr`. As a simple initial example, consider

14
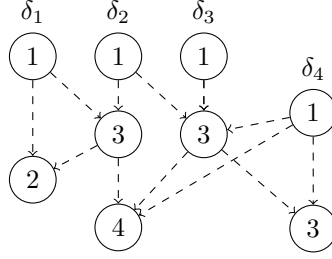
Figure 13: Example field of neighbour counting. In this example, the assumption is that neighbours are discovered upon send and reception of messages (for sends, the neighbours are visible since the next local round).

```
foldhood(0)(_ + _){ 1 }
```

which counts the number of neighbours at each device, possibly changing over time if the network topology is dynamic. The neighbourhood also includes the device itself. Indeed, in this example, folding collects the result of the evaluation of 1 against all neighbours, which simply yields 1 per neighbour. So, the effect is merely the addition of 1 for each existing neighbour—as we use the binary addition `_+_` as the accumulator function, of the accumulation value and the current, accumulating value. Note that the concern of neighbour discovery is orthogonal to the field program. In general, indeed, the neighbourhoods are managed by the aggregate computing platform. An example of system evolution is given in Figure 13.

Construct `nbr{e}` enables to "look around" just one step beyond a given locality, for what neighbours evaluate for `e`—or, dually, this models a device willing to advertise its view of `e` in its 1-hop surroundings. So, expression

```
foldhood(0)(_ + _){ nbr { sense[Double]("temperature") } } /
  foldhood(0)(_ + _){ 1 }
```

evaluates to the field of average temperature that each device can perceive in its neighbourhood. The numerator sums temperatures sensed by neighbours (or, analogously, it sums the neighbour evaluation of the temperature sensor query expression), while the denominator counts neighbours as described above.

Ordinary Scala functions can be defined to capture and give a name to common field computation idioms, patterns, and domain-specific operations. For instance, consider a `mux` function that implements a strictly-evaluated version of `if`:

```
def mux[A, B<:A, C<:A](cond: Boolean)(th: B)(el: C): A =
  if(cond) th else el
```

The *then* and *else* expressions are both evaluated first, and the proper result is then selected according to the Boolean condition With `mux`, a

15

```
def gradient(source: Boolean, metric: () => Double = nbrRange): Double =
  rep(Double.PositiveInfinity){ distance =>
    mux(source) { // Source devices yield 0.0
      0.0
    }{ // Others minimise over neighbours' gradient value + distance
      foldhoodPlus(Double.PositiveInfinity)(Math.min(_,_)){
        nbr{distance} + metric()
} } }
```

Figure 14: Simple gradient implementation.

variation of `foldhood`, called `foldhoodPlus`[4], which does not take the current device, or "self", into account, can be implemented as follows:

```
def foldhoodPlus[A](init: => A)(aggr: (A, A) => A)(expr: => A): A =
  foldhood(init)(aggr)(mux(mid==nbr{mid}){ init }{ expr })
```

Notice that the identity value `init` is used when considering a neighbour device whose identifier, `nbr{mid}`, is the same as that of the current device, `mid`, hence ensuring that `expr`, though evaluated, is *not* accumulated for "self". As another example, one can give a label to particular sensor queries, such as:

```
def temperature = sense[Double]("temperature")
def nbrRange = nbrvar[Double]("nbr-range")
```

The second case uses construct `nbrvar`, which is a neighbouring sensor query operator providing, for each device, a sensor value for each corresponding neighbour. For instance, for `nbrRange`, the output value is a floating-point number expressing the estimation of the distance from the currently executing device to that neighbour—so, it is usually adopted as a metric for "spatial algorithms".

Based on the above basic expressions, one can define a rather versatile and reusable building block of Aggregate Programming, called *gradient* [24, 25]. A gradient is a numerical field, exemplified in Figure 15, expressing the minimum distance, according to a certain `metric`, from any device to `source` devices; it is also interpretable as a surface whose "slope" is directed towards a given source. In SCAFI, it can be programmed as shown in Figure 14. The `rep` construct keeps track of the gradient values across rounds of computations: source devices are at a null distance from themselves, and the other devices take the minimum value among those of neighbours increased by the corresponding estimated distances as given by `metric`—defaulting to `nbrRange`. Notice that `foldHoodPlus` must be used to prevent devices from getting stuck to low values because of self-messages, as it would happen when a source node gets deactivated: with it, gradients dynamically adapt to changes in network topology or position/number of sources, i.e., they are self-stabilising [23].

Another common and important operation on fields is splitting computation

---

[4]The "Plus" suffix is to mimic the mathematical syntax R^+ of the transitive closure of a
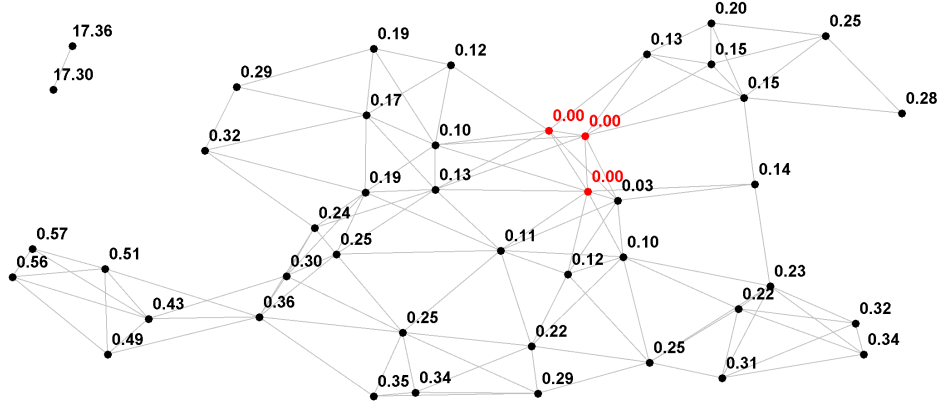
Figure 15: Pictorial representation of a gradient field snapshot in the midst of a simulation in SCAFI. The red nodes are the sources of the gradient. The nodes at the top-left have parted from the network and their values increase unboundly. The grey lines represent device connectivity according to a proximity-based neighbouring relationship.

into completely separate parts or sub-computations executed in isolated space-time regions. An example is computing a `gradient` in a space that includes obstacle nodes so that gradient slopes circumvent the obstacles. The technical issue, here, is to prevent obstacle nodes to participate in gradient construction and share a distance that could be wrongly selected by some device. Construct `branch(c){e1}{e2}` partitions the domain according to boolean field `c`: devices for which `c` is `true` run `e1`, others run `e`. Therefore, a gradient overcoming an obstacle is properly written as

```
val isObstacle = temperature > CRITICAL_TEMP // bool field of obstacles
branch(isObstacle){ Double.PositiveInfinity }{ gradient(isSource) }
```

since the `gradient` is effectively called and executed only by the set of devices for which `isObstacle` is `false`. We remark that the above field calculus expression of a gradient avoiding obstacles effectively creates a distributed data structure that is rigorously self-adaptive [23]: independently of the shape and dynamics of obstacle areas, source areas, metric and network structure, it will continuously bring about formation of the correct gradient, until eventually stabilising to it.

Such a behaviour is an example of collective intelligence at the edge: all the situated devices of the system continuously sense their local context and interact with neighbours to adjust their corresponding gradient value. Such a value, indeed, can be used to support services like: navigation of agents to a particular destination while avoiding risky areas (e.g., we can identify as obstacles those devices nearby overcrowded or dangerous situations), or providing directions for

---

(neighbouring) relation R.

broadcasting or collecting information in a network (see, e.g., the case study of Section 6.1) while excluding unreliable devices.

4. Extending Field Calculus with Concurrency: Aggregate Process Implementation in ScaFi

The field calculus is *space-time universal* [26] and, as such, is able to express every causal and Turing-computable distributed function. However, just like Turing-equivalence, this ability does not say much about how easy it is to encode a target behaviour, how understandable the resulting program will be, or how programs can be composed to scale with complexity. Relative expressiveness tests could be performed through compositional language embeddings, as studied in [27] for process algebras and in [28] for Linda dialects, expressed as process algebras following [29]. These tests can provide an answer to the problem of "practical expressiveness", however, they are only possible between languages with a similar model and structure. In particular, a comparison between process algebras and field calculus cannot be performed that way, and introducing notions of relative expressiveness for field calculi is left as future work.

In Section 4.1, we discuss the expressiveness limitations of field calculus, which revolves around the expression of *concurrency* within field computations. In this perspective, aggregate processes do extend the practical expressiveness of the field calculus, similarly to how concurrency mechanisms for multi-threaded programming extend the ability of traditional languages to express complex systems. Then, in Section 4.2, we enhance the field calculus, and hence the ScaFi reference implementation we use here, with a small set of mechanisms and library components to properly raise the abstraction level up to conveniently capture our notion of aggregate process. Our implementation is based on three pillars:

1. a system of processes is modelled as a "distributed field of sets of process descriptors", mapping each device to the set of processes it locally executes;
2. the computational activity run by an aggregate process is defined in terms of a field computation (namely, as the continuous evolution of a distributed data structure); and
3. the building block, called `spawn`, is added to allow programmers to define aggregate processes, generate instances, and manipulate their inputs and outputs—i.e., essentially specifying what collective activities are to be spawned, when and where, and how they relate within the overall aggregate program.

4.1. Alignment and Dynamic Field Expressions

Results of field computations, at runtime, can be represented by hierarchical structures known as *value-trees (vtrees)*. A vtree is an ordered tree of values tracking the result of any evaluated sub-expression. The vtree θ produced by

```
branch(isObstacle){ Double.PositiveInfinity }{ gradient(isSource) }
```

(a) Gradient computation around obstacles.
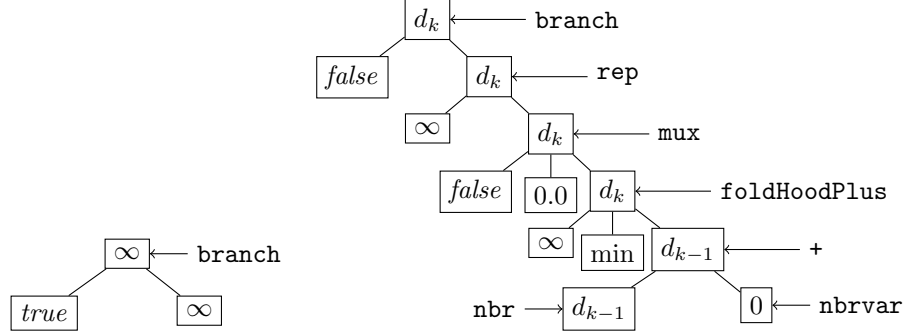
(b) Vtree computed by obstacle nodes.

(c) Vtree computed by non-obstacle nodes.

Figure 16: Different kinds of vtrees for a gradient computation around obstacles.

the evaluation of an expression e is denoted as $v\langle\theta_1,\ldots,\theta_n\rangle$, where the root v is the value produced by e, and $\theta_1,\ldots,\theta_n$ are the vtrees produced by the immediate sub-expressions of e. The operational semantics of the field calculus leverages vtrees; in other words, an evaluation of field expressions is a process building a vtree. So, essentially, the output of an execution of an aggregate program is a vtree. Notice that two devices of the same aggregate may yield vtrees with *different shape* (i.e., different structure beside different values for the same nodes) while evaluating the same aggregate program, e.g., due to branching constructs. Fundamental to the machinery and compositionality of the approach is the notion of *alignment* [30], by which evaluation (construction of vtrees) is defined in terms of other structurally-equivalent vtrees: the vtree corresponding to the previous round of the executing device, and the vtrees of neighbours devices. When two vtrees are not structurally-equivalent (i.e., they have different nodes), they do not "align", and hence one cannot be used with the other; notice, however, that two vtrees may *partially* align, and hence interaction is possible only within the aligned subtrees.

As already mentioned in Section 3, the local execution of a field computation yields an *export* message that is meant to be sent to neighbours in order to sustain the global behaviour of the aggregate. Such an export can also be seen as the state- and communication-related part of vtrees.

For instance, the expression of Figure 16a yields the following vtrees:

- For obstacle nodes, the vtree of Figure 16b. The root of the vtree, ∞, is the result of the whole expression. The left sub-tree is the result of the first sub-expression: isObstacle, which evaluates to *true*). Finally, the right sub-tree is the result of the branch taken: Double.PositiveInfinity, which evaluates to ∞.

19

- For non-obstacle, non-source nodes, the vtree of Figure 16c—where d_k is the current distance estimate, d_{k-1} the previous one, and the main sub-expressions relative to nodes are reported on their right. For source nodes the vtree is the same, except that $d_k = 0$.

Notice that both state and communication are based on alignment: `rep` retrieves the value to work on from the previous vtree of the device, and `nbr` gets the values from neighbours by observing the nodes in the corresponding vtrees that have the same place in the computation as the current vtree node. In other words, interaction works on a structural basis where order matters.

However, dynamicity – due to potentially different and unknown activities – would break ordering and hence alignment, possibly leading to ambiguous or inconsistent vtree entries.

4.1.1. Paradigmatic Example of the Alignment Issue: Limited Multi-Gradient

The `gradient` function already supports the creation of a gradient from multiple sources: it is sufficient to build a source input field that is true in correspondence of multiple nodes.

```
val isSource = sense[Boolean]("source")
gradient(isSource)
```

With this approach, however, the resulting gradient field could not be used, e.g., to collect information into a source from nodes that are beyond the midpoint between that source and another adjacent source. The solution is to leverage multiple independent gradient computations that can overlap within the aggregate. For a fixed number of gradients, the following code works.

```
(gradient(source1), gradient(source2), ...)
```

However, there is no trivial way to handle a dynamic number of gradients that are to be generated and destroyed as new sources activate or deactivate, respectively, without breaking alignment. For instance, the following code

```
val sources: Set[ID] = // gossip the source set
val gradients: Map[ID,Double] =
  sources.map(source => source -> gradient(source==mid)).toMap
```

would break in unexpected ways.

4.1.2. Solution: Alignment over Arbitrary Keys

The previous example involves evaluating a field expression in an iterative context. When mapping a dynamic collection, the number of elements and the order of traversal do not allow for drawing a consistent correspondence between two iteration steps of two devices. This unless one manually introduces keys or tags for the field expressions, e.g., based the identity of the mapped elements. Therefore, we address this problem by the *primitive mechanism*, called `align`, with signature

```
def align[K,V](key: K)(proc: K => V): V
```

to enable alignment on arbitrary keys, namely, to introduce a new computation scope by inserting a vtree node tagged with the provided key. Upon this, the previous code can be fixed as follows:

```
val gradients: Map[ID,Double] =
  sources.map(source =>
    source -> align(source){ _ => gradient(source==mid) }
  ).toMap
```

However, this approach is quite low-level, and does not properly handle lifecycle, which is currently expressed by field `sources`, e.g., assumed to be provided by means of gossiping. Therefore, we use the principle explained in this section to define a more expressive construct that provides both aligned execution and automatic propagation of keys. Such construct, `spawn`, effectively provides an implementation of our aggregate process abstraction.

4.2. Aggregate Processes in SCAFI

The `spawn` primitive supports our notion of aggregate processes by handling activation, propagation, merging, and disposal of process instances for a specified kind of process. Coherently with the formalisation in [21], it has signature:

```
def spawn[K,A,R](process: K => A => (R,Boolean),
                 newKeys: Set[K],
                 args: A): Map[K,R]
```

It is a generic function, parametrised by three types:

K — the type of process *keys*;

A — the type of process *arguments*, or inputs;

R — the type of process *result*.

The function accepts three formal parameters:

1. `process` — has type `K=>A=>(R,Boolean)` and expresses the computation logic of the process by a curried function taking a key, an argument, and then returning a pair of the computation result and a boolean *status* value expressing whether the current device is willing to participate in the process instance or not;
2. `newKeys` — is the set of keys of the processes to be spawned; and
3. `args` — is the "runtime argument" for the process instances active in this round.

Remember that values are fields—e.g., `newKeys` is a field of sets which may have entries only in specific devices and execution rounds, and `args` is a field whose values of type `A` may differ in different space-time locations. By a local

21

```
val vm: RoundVM = // provides access to virtual machine calls

def spawn[K, A, R](
    process: K => A => (R, Boolean),
    newKeys: Set[K],
    args: A
): Map[A,R] = {
  rep(Map[K, R]()) {
    case processMap => {
      // 1. Take active process instances from my neighbours
      val nbrProcs = includingSelf.unionHoodSet(nbr{ processMap }.keySet)
      // 2. New processes to be spawn
      val newProcs = newKeys
      // 3. Collect all process instances to be executed,
      //    execute them, and update their state
      (nbrProcs ++ newProcs).map { p =>
        vm.newExportStack
        val result = align(puid) { _ => process(p)(args) }
        // Discard the export of the previous step if status is false
        if(result._2) vm.mergeExport else vm.discardExport
        p -> result
      }
      .collect { case(p,pi) if pi._2 => p -> pi._1 }
      .toMap
    }
  }
}
```

Figure 17: Illustrative implementation of `spawn` in ScaFi

perspective, **spawn** accepts a *set* of keys to allow *generation* of zero or more process instances at the device in the current round. Note that a process key has a twofold role: it works both as a *process identifier* (PID) and as *initialisation* or *construction parameter*. When different construction parameters should result in different process instances, it is sufficient to instantiate type K with a data structure type including both pieces of information and with proper equality semantics. Notice that if a new key already belongs to the set of active processes, there will be no actual generation, or restart, but *merging* instead, since identity is the same as an existing process instance. Finally, note also that the outcome of **spawn**, namely, a map from process keys to process result values, can in turn be used to fork other process instances or as input for other processes. Indeed, the basic means for processes to interact is to connect the corresponding **spawn**s with data.

4.3. Behind-the-Scenes: *spawn* Implementation

To provide an intuition of the operational semantics of aggregate processes, formalised in [21], we take a look at the implementation of **spawn**, illustrated

in the listing of Figure 17. Abstracting from ancillary details, `spawn` internally works as follows:

1. it combines new process keys with previous ones from the device itself and those from direct neighbours,
2. maps the resulting keyset by running `process` in an aligned way w.r.t. the process keys; and finally
3. filters results upon the boolean status value.

Crucially, the filtering of results, achieved through the `vm` calls, prevents the writing of exports: so, filtered processes are not broadcast to neighbours. This mechanism ultimately impacts the spatiotemporal evolution of a process.

4.4. Support for Aggregate Processes

The aggregate approach and its SCAFI embedding covered in Section 3 do support the requirements presented in Section 2.1 through the extensions covered in this section. Indeed:

- they foster a collective stance because the behaviour of an ensemble is expressed by a global perspective and reified as aggregate computations executed repeatedly and in concert by every device participating in the system [16];

- the field abstraction and neighbour-based interaction model naturally support dynamic and contextual responses to network and environment change [23];

- resiliency is promoted by the self-organising and eventual consistency properties of aggregate computing [16];

- opportunism in resource exploitation is enabled by the flexibility in the execution and deployment of aggregate systems [22];

- integration with mainstream paradigms, techniques, and tools is supported by the SCAFI DSL embedding into the Scala language [31].

In Section 5, we cover in detail how the process features shown in Figure 2 and implied by the described implementation can be specified, programmatically.

5. Programming with Aggregate Processes: Techniques and Patterns

In the following, we discuss programming and management of aggregate processes activated through `spawn`. We start from the basics (process definition, lifecycle and boundary management) and then introduce more complex examples in order to delineate the principle behind an "aggregate process API", as well as to prepare for the case studies that follow—concretely showing how composition of collective behaviour could support the engineering of pervasive applications.

5.1. Process Definition

Defining a type of process merely consists of defining a function that can be passed as the `process` parameter to `spawn`. It must be a curried function from a process key `K`, an argument `A`, to a tuple result `(R,Boolean)`—for some choice of `K`, `A`, `R` made statically at a particular call of `spawn`.

It is good practice to define custom types for `K`, `A`, and `R`, e.g.:

```scala
case class PID(id: Int)(val initiator: ID)
case class PArgument(arg: Int)
case class PResult(result: String)
```

Therefore, a process definition could take the following schema:

```scala
// Method syntax
def myProcessLogic(pid: PID)(parg: PArgument): (PResult,Boolean) = {
  val result: PResult = ??? // compute result
  val stay: Boolean = ??? // compute logic for process boundary/lifecycle
  (result, stay) // returned pair
}

// Function syntax
val myProcess = (pid: PID) => (parg: PArgument) => ???

// or, from a method:
val myProcess: PID => PArgument => PResult = myProcessLogic _
```

Once we have defined a process function, we can use `spawn` to create process instances:

```scala
spawn[PID,PArgument,PResult](myProcess, ...)
```

5.2. Process Generation (Lifecycle Management part 1/2)

Generating process instances is just a matter of creating a field of keysets that become non-empty as soon as some "triggering" space-time event has been recognised. Examples include spatial conditions on sensors data and computation, timers firing, and so on [23].

Consider the following simple process definition:

```scala
type K = Int // The type (alias) of process keys
type A = Int // The type (alias) of process arguments
type R = (Int, Boolean) // The type (alias) of process return

def m(k: K)(a: A): R = (k + a, true) // true means: always participate
val p = m _
```

A trivial example could leverage a constant, uniform field with full domain:

```scala
val keySet = Set(1)
val argument = 2
val processes = spawn(p, keySet, argument)
```

In this case, a single process instance gets activated everywhere, and repeatedly appliedm on a round by round basis, against a constant argument: for every device (everywhere), `processes` is always (everytime) a `Map(1->3)`. Of course, we can spawn multiple instances of the same process type in a single `spawn`, and provide a non-uniform argument field. For instance, expression

```
// Remember: mid is the field of local device IDs
spawn(p, newKeys = Set(1,2), args = mid)
```

yields a constant field that is locally `Map(1->1+`δ`,2->2+`δ`)` for any device δ.

Things get more interesting when the keyset field is non-uniform. Consider a connected system of three devices δ_1, δ_2, δ_3. Since process keys are automatically propagated to neighbour devices, expression

```
spawn(p, newKeys = mid, args = 0)
```

will stabilise to a field that is everywhere `Map(`δ_1`->`δ_1`,`δ_2`->`δ_2`,`δ_3`->`δ_3`)`. In this case, the"source" or "generator" of process with PID δ_i is the device δ_i itself. The time it takes for a process to spread depends on the timing of round execution and communication acts in the different devices. Now, suppose the system gets split into two partitions (δ_1, δ_2) and (δ_3), and that, later, the latter is joined by a device δ_4: under these circumstances, the output will remain the same for δ_1, δ_2 whereas δ_3 and δ_4 will both output `Map(`δ_1`->`δ_1`,`δ_2`->`δ_2`,`δ_3`->`δ_3`,`δ_4`->`δ_4`)`.

Typically, processes are generated by specific devices, when specific conditions come true. This is modelled by a keyset field which is empty everywhere in space-time except in locations where the event is recognised. A schema is as follows:

```
val event: Boolean       = // ...
val generateKey: Any => K = // ...
val keys: Set[K] = if(event){ Set(generateKey(???)) } else { Set.empty }

spawn(???, keys, ???)
```

you generally need a way to generate a process key to uniquely identify a process instance with the particular occurrence of the event.

As mentioned before, process keys work both as process identities and as construction parameters. Consider this process modelling a gradient computation:

```
def gradientProcess(source: ID)(obstacle: Boolean): Double =
  branch(!obstacle){ gradient(source==mid) }{ Double.PositiveInfinity }
```

In this case, since the `ID` of the source is used to identify a process instance, you cannot have more than one gradient process per source. Now, suppose you want to preserve the same semantics but also keep track of the device who generated the process (which is not necessarily the source of the gradient): you do not want process identity to depend on the generator, so your key data type must carry the additional information while handling identity (i.e., equality) like in

the previous example. For this purpose, the following Scala `case class` idiom comes handy:

```scala
case class PID(source: ID)(val generator: ID)
```

Finally, a clarification is needed, regarding the semantics of `spawn` and the peculiar execution model of round-by-round field computations, as they especially relate to branching. Construct `spawn` differs from traditional "thread spawning" constructs like Erlang's `spawn` or Java's `Thread.start()`, in that a SCAFI's `spawn` expression needs to always be evaluated in each round in order to carry through active process instances. That is, in the following program,

```scala
branch(someCondition){
  spawn(???, ???, ???)
}{
  spawn(???, ???, ???)
}
```

taking a branch will cause the destruction of all the process instances `spawn`ed in the other branch.

5.2.1. Time tracking in SCAFI

Basic techniques for process generation include space-time event recognition and time-wise scheduling. For the purpose, building block `T` is used to model the passing of time in field computations [23]. In SCAFI, it can be implemented as follows.

```scala
def T(init: Double, floor: Double, decay: Double => Double): Double =
  rep(init) { v => Math.min(init, Math.max(floor, decay(v))) }

def T(init: Double, delta: Double): Double =
  T(init, 0.0, (t: Double) => t - delta)
```

Operator `T` works by keeping track of the remaining time (starting from `init`) via construct `rep`, and then using the provided function `decay` to enact the passing of time until `floor` is hit. A derived version based on a `delta` step can be straightforwardly defined. Built-in, local sensor `dt()` is used to locally keep track of time passed since the previous computation round. Using `T`, it is trivial to spawn a process once after some delay:

```scala
val newPids = mux(T(100, dt())==0){/* gen new keyset */}{ Set() }
```

The key thing to understand is that such a "once timer" restarts any time the corresponding computation is "re-entered". In other words, it is refreshed when the corresponding computation is not executed, since its `rep` node, by disappearing from the vtree, loses its memory; hence, a clock based on a cyclic timer can be implemented as follows.

```scala
def clock(len: Long, decay: Long): Long =
  rep((0L,len)){ case (k,left) => // Function defined by pattern matching
```

```
    branch (left == 0){ (k+1,len) }{ (k, T(len, decay).toLong) }
 }._1 // "_1" projects to the first element of the tuple
```

Such a clock can be used for periodically spawning processes: see, e.g., the `replicated` example below.

5.3. Process Expansion/Shrinking (Boundary Management)

Notice that a condition for process generation is that the generating device does not immediately quit itself. By `spawn`, every process instance is *automatically propagated by all the participating devices to their neighbours.* Such a propagation does not occur only if the device returns status `false`—meaning that the device does not want to participate in that process instance. Therefore, it is possible to regulate the shape of such "computational bubble" by dictating conditions by which a device must return status `false` (i.e., meaning *external* to the bubble)—as mentioned, this indicates the willingness to *stop* computing, or participation in the process. That is, only devices that return status `true` are *internal* and so will propagate the process.

Moreover, such a propagation happens continuously: so, a device that exited from a process may execute it again in the future, if any of its neighbours is still internal to that process. In particular, the *border* of a process bubble is given by the set of all the devices that are external but have at least one neighbour which is internal. As long as a node is in the border, it continuously re-acquires and immediately quits from the process instance: this *continuous evaluation of the border* is what ultimately enables a spatial *expansion* of the process bubble. Conversely, a process bubble gets restricted when internal nodes become external; this is called *shrinking*.

As an example of expansion and shrinking, consider the system evolution of Figure 18, where a process instance is generated at δ_1 and $\top$ (resp. $\bot$) represents `true` (resp. `false`) status.

5.4. Process Termination (Lifecycle Management part 2/2)

As we have seen, a process instance *terminates* when all the devices quit by returning status `false`. Implementing process termination may not be trivial, since proper local or global conditions must be defined so that the "collapsing force" can overtake the "propagation force". Precautions should be taken so that external devices do not re-acquire the process: the border should steadily shrink, also considering temporary network partitions and transient recoverable failures from devices. In the following pages, we will develop an higher-level support to process termination based on "termination signals".

Example: spatiotemporally limited processes. It is often useful to run processes on a limited subset of the devices (e.g., those contained within a certain range from the process generator), for a limited amount of time. In order to support this, a process should carry information about the generation location, the distance from the generation location, the time of generation, and the time that

27
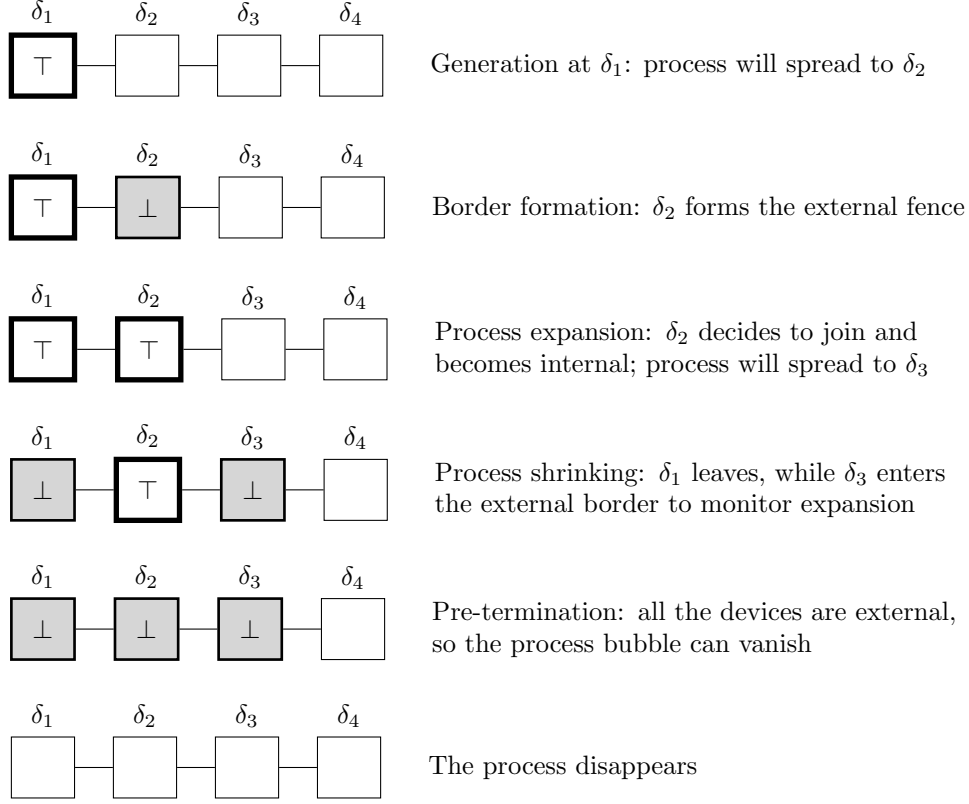
Figure 18: Example of a system evolution comprising all the dynamic phases (generation, expansion, shrinking, and termination) of an aggregate process instance. Each line shows the network at a given time instant and the corresponding output status for a single aggregate process instance. Nodes with bold border denote devices participating in the process with `true` status; grey nodes denote devices of the border with `false` status; empty nodes do not evaluate the process at all.

has elapsed since generation. Border and lifecycle management should manage and predicate on such information. In the example of Figure 19, devices call themselves out when the process time exceeds `lifetime` or the distance computed by the gradient exceeds `maxRange`.

Notice that, in circumstances like this, the logic of computation can be completely separated from process border and lifecycle management; in these cases, program design can benefit from *separation of concerns*, adopting a *single-responsibility principle* while functional composition enables creation of a full process definition for `spawn`. Moreover, with careful design, this enables *reusability* of lifecycle strategies, as shown in Figure 20. The `STLifecycle` can be combined with *any* process logic over process keys that conform to

```
case class PID(pid: String)
              (val generator: ID, val startTime,
               val lifetime: Long, val maxRange: Double)

type K = PID
type A = Unit // we are not interested in any runtime argument
type R = Int

def logic(k: K)(a: A): R = 0 // trivial
def lifecycle(k: K)(a: A): Boolean =
  time()-k.startTime < k.lifetime &
    gradient(k.generator==mid) <= k.maxRange

// This utility function merges logic with lifecycle functions into one
def combine[K,A,R1,R2](f1:K=>A=>R1)(f2:K=>A=>R2):K=>A=>(R1,R2) =
  k => a => (f1(k)(a), f2(k)(a))

spawn[K,A,R](combine(logic)(lifecycle), /* newKeys */, ())
```

Figure 19: Example of a spatiotemporally limited process.

STLimitedProcessKey[5].

5.5. Process Abstraction

Using functional abstraction, it is possible to define high-level behaviours that provide a clean interface hiding the complexity of internal process management.

Example: time replication. In [33], a technique based on time replication for improving the dynamics of gossip is presented. It works by keeping k running replicates of a gossip computation executing concurrently, each alive for a certain amount of time. New instances are activated with interval p, staggered in time. The whole computation always returns the result of the oldest active replicate. This is intended to improve the dynamics of algorithms, providing an intrinsic refresh mechanism that smoothly propagates to the output. With spawn, it is trivial to design a replicated function that provides process replication in time.

```
def replicated[A,R](proc: A => R)(argument: A, p: Double, k: Int) = {
  val lastPid = clock(p, dt())
  spawn[Long,A,R](pid => arg => (proc(arg), pid > lastPid+k),
                  Set(lastPid), argument)
} // returns a Map[Long,R] from replicate IDs to corresponding values
```

---

[5]Notice that Scala provides mechanisms, such as *structural types* or the *pimp-my-library pattern* [32], to avoid the requirement of explicit, apriori trait implementation.

```
trait STLimitedProcessKey {
  val generator: ID
  val startTime: Long
  val lifetime: Long
  val maxRange: Double
}

def STLifecycle[K,A,R](k: K <: STLimitedProcessKey)(a: A): Boolean =
  time()-k.startTime < k.lifetime &
    gradient(k.generator==mid) <= k.maxRange
```

Figure 20: Schema of a pattern for creating reusable lifecycle strategies.

```
case class Msg[V,From,To](body: V, from: From, to: To)
type MBox = List[Any]
type PostOffice = Map[Any,MBox]

rep[PostOffice](Map.empty)(msgs => {
  spawn(???, ???, Args1(???, msgs))
  spawn(???, ???, Args2(???, msgs))
  msgs
})
```

Figure 21: Process interaction idiom: using `rep` to support interaction of processes spawned in different parts of the source code.

clock is a distributed time-aware counter [33] yielding an increasing number i at each interval p that represents the PID of the i-th replica. Notably, in this case, every device can locally determine when it must quit a process instance. Moreover, the exit condition based on PID numbering, `pid > lastPid+k`, prevents process reentrance. Section 6.2 provides an empirical evaluation of the behaviour of function `replicated`.

5.6. Process Interaction

The most basic means to make aggregate processes interact is by *piping* the output of a process into the input of another.

```
val p1s: Map[K1,R1] = spawn[K1,A1,R1](???, ???, ???)
val arg: R1 = p1s.headOption.getOrElse(??? /* some default value */)
type A2 = R1
val p2s: Map[K2,R2] = spawn[K2,A2,R2](???, ???, arg)
```

Moreover, the programming idiom of Figure 21 can be used in the case of mutually feeding processes, `spawn`s in different scopes, or when a "program-wide" communication structure is desired.

```scala
trait Status

case object ExternalStatus extends Status    // External to the bubble
case object BubbleStatus extends Status       // Within the bubble
case object OutputStatus extends Status        // Within the bubble + output
case object TerminatedStatus extends Status // Willingness to shutdown

val External: Status = ExternalStatus
val Bubble: Status = BubbleStatus
val Output: Status = OutputStatus
val Terminated: Status = TerminatedStatus

case class POut[T](result: T, status: Status)
object POut {
  // Implicit definition to map POut to (T,Boolean)
  implicit def toBasicSpawnTuple[T](pout: POut[T]): (T,Boolean) =
    (pout.result, pout.status!=External)
  // Conversion between process computation definitions
  implicit def fto[K,A,R](proc: K => A => POut[R]): K=>A=>(R,Boolean) =
    k => a => toBasicSpawnTuple(proc(k)(a))
}
```

Figure 22: Process statuses.

5.7. More Expressive Process Definitions

Now, we show how to support more declarative process definitions by leveraging expressive "statuses", modelled in Figure 22. First, we define the concrete `Status`es. We also capture a process output not as a tuple `(T,Boolean)` but as a tuple `(T,Status)`, which we render as an algebraic data type `POut` to provide useful implicit conversions to the former form (leveraging the power of Scala). At this point, we can handle termination as per Figure 23 by mapping process computation definitions. We can employ a simple shutdown algorithm that distributes the termination signal to neighbours, closes the process in a device, by going `External`, when all the neighbours have received such a signal, and prevents re-acquisition of the process if any neighbour presents the termination signal.

Output filtering is achieved by mapping results to optional values that are present only when the device has status `Output`; however, this also requires a filtering outside the call to `spawn`. Function `handleOutput` in Figure 24 maps process results, of type `T`, to `Option[T]` values, present (constructor `Some`) or not (constructor `None`) based on whether status is `Output` or not, respectively.

Finally, a higher-level "spawn" function `statusSpawn` can be defined as per Figure 25, where `handleOutput` and `handleTermination` wrap the given `process` (which must yield a `POut[T]` value), and only `Option[R]` values that are present (not `None`) are kept. See Figure 26 for a graphical example.

```scala
def handleTermination[T](out: POut[T]): POut[T] = {
  rep[(Boolean,Int,POut[T])]((false,0,out)){
    (terminated,k,res) =>
      val mustTerminate = out.status==Terminated |
                          includingSelf.anyHood(nbr{terminated})
      val mustExit = includingSelf.everyHood(nbr{mustTerminate})
      (mustTerminate, // true if observed termination signal
      1, // flag (k=0 only in the first round for this process)
      if(mustExit || (mustTerminate && k==0))
        (out.result, External) // enforce quit
      else
        out // just pass given (output,status) through
      )
  }._3
}
```

Figure 23: An example of a process termination algorithm.

```scala
def handleOutput[T](out: POut[T]): POut[Option[T]] = out match {
  case POut(res, Output) => POut(Some(res), Output)
  case POut(_, s) => POut(None, s)
}
```

Figure 24: A function for filtering process outputs according to process statuses.

Example: limited multi-gradient. The problem described in Section 4.1 of activating a spatially-limited gradient computation for each device where sensor `isSrc` gives true, and deactivating it when it stops doing so, can be solved as per Figure 27. There, we also show the "closure idiom", by which a process behaviour is defined as a *closure*, i.e., a function closing over its environment (in this case, parameter `isSrc`).

6. Case Studies

In this section, we exercise the constructs previously introduced by presenting three application examples. The first two case studies, originally presented in [21], are described here in more detail, while the third one is a novel contribution addressing the edge-cloud domain.

We implemented all the scenarios in SCAFI combined with the Alchemist simulator [34], described below. Data generated by the simulator has been analysed using xarray [35]; visual reports of the data have been created via Seaborn and matplotlib [36]. For every experiment, we execute 101 runs; results reported in this manuscript represent their average. For the sake of reproducibility, the source code and instructions for running experiments are publicly available in two separate repositories: one hosting the former two

```
def statusSpawn[K, A, R](process: K => A => POut[R],
                         newKeys: Set[K],
                         args: A): Map[K,R] =
  spawn[K,A,Option[R]](
    k => a => handleOutput(handleTermination(process(k)(a))),
    params,
    args).collectValues { case Some(p) => p }
```

Figure 25: More expressive process spawning.

case studies[6]; and a separate one for the latter[7], there including additional implementation details.

A key goal of these case studies is to demonstrate the soundness of our aggregate process implementation. Moreover, our empirical evaluation will also show that, orderly: *(i)* in certain cases, aggregate processes can greatly limit the consumption of computational resources while retaining a reasonable quality of service (QoS); *(ii)* in certain cases, powerful meta-algorithms enabled by aggregate processes can improve the dynamics of distributed computations. Moreover, we also intend to show how aggregate computing and processes promote the implementation of systems exhibiting forms of collective intelligence. In particular:

1. in the messaging case study, Section 6.1, the aggregate is able to create dynamic *teams* of devices in order to hop-by-hop connect a source of a message with the corresponding recipient, through a central, coordinator node;
2. in the swarm case study, Section 6.2, the drones exploring the environment in a mission collaborate to agree on the risk of fire;
3. in the edge computing case study, Section 6.3, the nodes in the infrastructure coordinate and adapt to provide efficient service discovery.

Simulation Framework

The Alchemist simulator has been selected as it features a meta-model, depicted in Figure 28, suitable for aggregate computing system models. In the simulator, *nodes* are situated inside an *environment* and are connected to each other through a *linking rule*. Nodes are programmed with *reactions*, which are rules guarded by *conditions* and causing *actions* on the environment. Reaction occurrence is described by an equation considering a *time distribution* and possibly the current state of the conditions. Nodes contain *molecules* (data identifiers), which are associated to *concentrations* (data values). The simulator leverages an extended version of the Gibson-Bruck kinetic Monte Carlo algorithm [37] to determine which event to process next, and efficiently deals

---

[6]https://bitbucket.org/metaphori/experiment-spawn
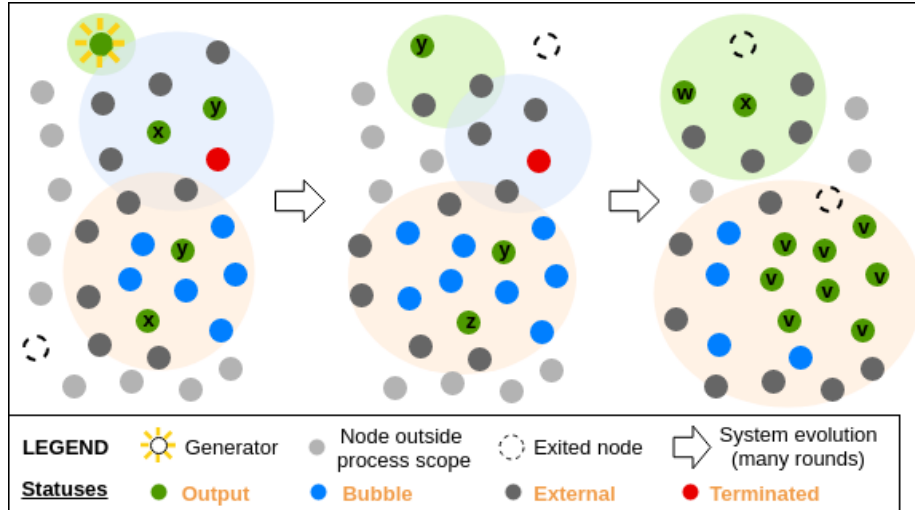[7]https://github.com/DanySK/Experiment-2019-EAAI-Processes

Figure 26: Graphical example of the evolution of a system of processes and the role of statuses in `statusSpawn`. The green bubble springs into existence; the blue bubble dissolves after termination is initiated by a node; the orange bubble expands. Only output nodes will yield a value. Bubbles may of course overlap (i.e., a node may participate, with different statuses, to multiple processes) and the dynamics can be arbitrarily complex (because of mobility, failures, and local decisions).

with chains of causal events by maintaining a dependency graph among reactions. In fact, Alchemist's concepts can be instanced as follow (cf. Figure 28 and Figure 6):

- *nodes* are devices executing an aggregate program;

- *linking rules* are leveraged to model neighbourhoods;

- *reactions* are exploited to model the proactive parts of the software, namely, the actual evaluation of the aggregate program and the subsequent sending of coordination messages;

- *molecules* keep state for devices, including, e.g., sensors and actuators identifiers, used to access the local device capabilities; and finally

- *concentrations* are the actual values read from sensors and sent to actuators.

Alchemist is a reference simulator for aggregate computing, leveraged in almost all its research activities requiring empirical evaluation by simulation [9, 16], and providing support for various aggregate computing languages, such as SCAFI [19] and Protelis [38] programs. Alchemist also provides integration with OpenStreetMap [39] and Graphhopper[8], enabling simulation of real-world

---

[8]https://www.graphhopper.com/

34

```
def multiGradient(isSrc: Boolean, maxExtension: Double) =
  statusSpawn[ID,Double,Double](src => limit =>
    gradient(src==mid,nbrRange) match { // consider the usual gradient
      case g if src==mid && !isSrc => (g, Terminated) // close on unsource
      case g if g>limit => (g, External)              // out of bubble
      case g => (g, Output)                           // in bubble + get
    },
    newKeys = if(isSrc) Set(mid) else Set.empty,
    args = maxExtension
  )
```

Figure 27: An example implementation of the limited multi-gradient functionality.

situated scenarios; and is equipped with a batch engine for automating the execution of several repetitions of a simulation.

6.1. Opportunistic Point-to-Point Instant Messaging

6.1.1. Motivation

The possibility of communicating by delivering messages regardless the presence of a conventional Internet access has recently gained attention as a mean to work around censorship[9] as well as in situations with limited access to the global network—e.g., in rural areas, in exploration of unknown territories, in emergency scenarios or during urban events when the network capability is overtaken.

Here, we consider a messaging application where a source device, or *sender*, wants to deliver a payload to a peer device, aka *recipient*, *target*, or *destination*, in a hop-by-hop fashion by exploiting nearby devices as relays. The source device only knows the identifier of the recipient, whose spatial and network locations are unknown, and no viable route is pre-determined. The recipient's identifier could be obtained either from an infrastructural bootstrapping process assigning and distributing identities (i.e., a registration process), or in a entirely distributed fashion through an aggregate predicate on the device state (i.e., a function computing a boolean field).

Our goal is to show how aggregate processes can support this kind of application featuring multiple concurrent messages. It will do so in a self-organising way, by limiting the number of devices involved in message delivery, and leading to bandwidth savings and energy savings in turn.

6.1.2. Opportunistic Chat: Aggregate Process-based Implementation

The idea of the case study is to activate an aggregate process instance for each message sent from a source node to a destination node, and to limit the extension of such process instance to a small subset of the devices belonging to
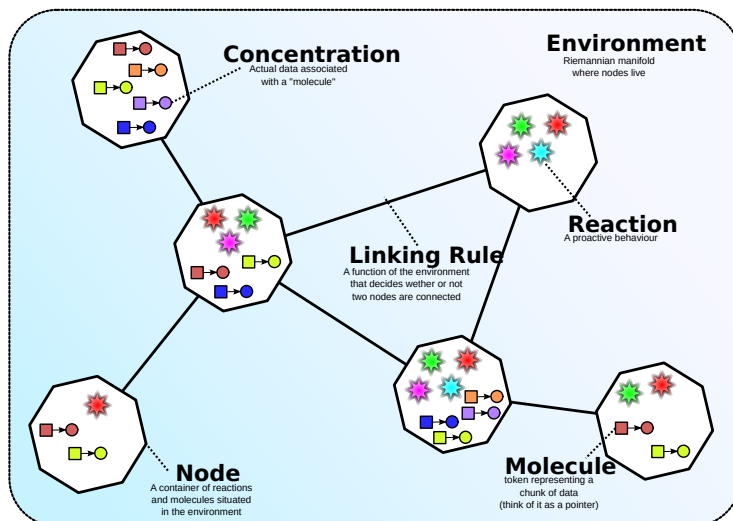
---

[9]http://archive.is/C3niO

35

Figure 28: A pictorial representation of the Alchemist simulator meta-model, taken from [34]. Aggregate programming concepts are mapped onto the simulator abstract model, producing an aggregate computing-enabled simulation platform, upon which the experiments have been executed.

the whole system. A simple algorithm to do so involves creating information flows from the source node to a "central" coordinator node, and from the central node to the recipient node. Once the recipient has received the message, the message delivery process must be closed.

An implementation can be as shown in Figure 30. First, we model the data, i.e., the message, which also represents the PID, and coordination data used for directing the shape of the process, which also represent the runtime arguments. Then, we define the process computation logic, where the target of the message, `msg.target`, has status `Output` at first and then `Terminated`. Indeed, after it has read the message, the process bubble can vanish. Only nodes for which field `inRegion` is locally true have status `Bubble` and hence participate in the message delivery process for `msg`. The nodes for which `inRegion` is true are those that belong to the path from the source of a message to the coordinator *or* to the path from the coordinator to the recipient; these are calculated, respectively, exploiting knowledge of nodes on the minimal path to the coordinator (`parentToCentre`) and knowledge of neighbours whose minimal path to the coordinator includes the current node (`dependentNodes`), in turn calculated from the knowledge of the parent device of the spanning tree implied by a gradient from the centre—cf. `gradientWithParent` call in the next listing. Notice that the code of `chatProcessLogic` actually deals only with the management of the process boundary and lifecycle; indeed, the "core logic" of such a process is merely the expression returning `msg` as output. Finally, we define a `chat` function that leverages `statusSpawn`, where the device used as centre and new messages to be sent are externally provided through parameters
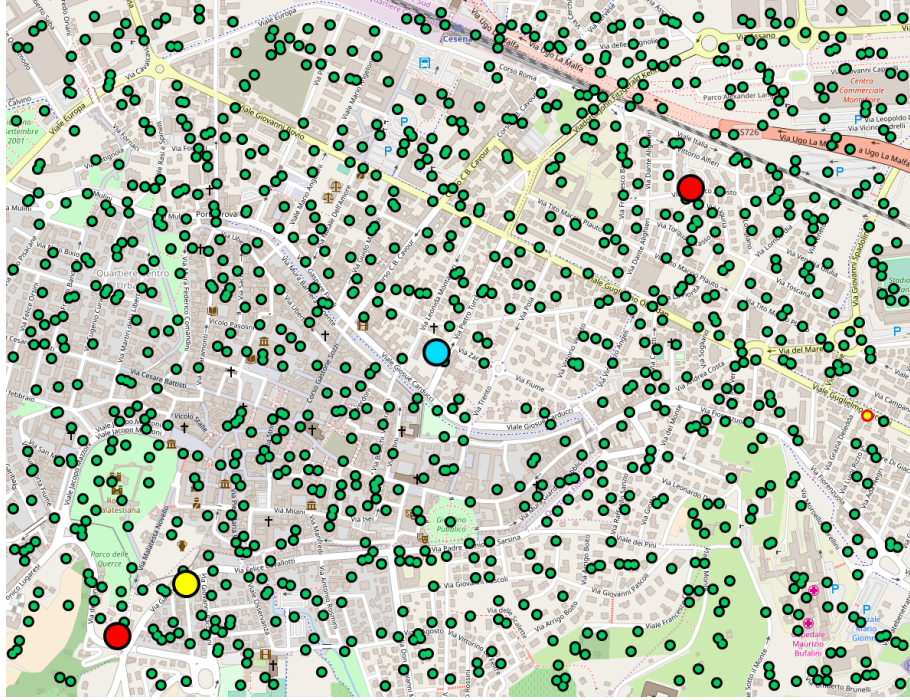
36

Figure 29: Snapshot of the chat users in the city of Cesena. Green dots are users participating the system, the big sized yellow dot is the coordinator, big sized red dots represent devices currently sending a new message, big sized blue dots represent devices currently targeted to receive a message.

`centre` and `newMsgs`, respectively. The output of function `chat` is the field of the collections of messages that have been currently received at the recipient devices.

6.1.3. Experimental Setup

We compare two aggregate implementations of such messaging system. The first implementation, called *flood chat*, simply broadcasts the payload to all neighbours. In spite of an in-place garbage collection system, however, this strategy may end up dispatching the message towards directions far-off the optimal path, burdening the network and memory capabilities of the system. The second implementation, *spawn chat*, leverages `spawn` in order to reduce the impact on the network infrastructure by electing a node as coordinator. Then it creates an aggregate process connecting the source with the coordinator and the coordinator with the target, and finally delivers the message along such support. In this experiment, we naively choose a coordinator randomly, but better strategies could be deployed to improve over this configuration [16], including a self-organising election of the centre, a better positioning of such dynamic centre along the shortest path, and improved resilience to network

```scala
case class Msg(src: ID, target: ID, str: String)
case class ChatArgs(parentToCentre: ID, dependentNodes: Set[ID])

def chatProcessLogic(msg: Msg)(args: ChatArgs): POut[Msg] = {
  // Boolean field denoting the path from the msg source to the centre
  val srcToCentrePath = msg.src == mid | includingSelf.anyHood {
    nbr(args.parentToCentre) == mid
  }
  // Boolean field denoting the path from the centre to the target
  val destToCentrePath = args.dependentNodes.has(msg.target)
  // Boolean field denoting the process domain (set of participants)
  val inRegion = srcToCentrePath || destToCentrePath
  POut(
    result = msg,
    status =
      branch (mid == msg.target) {
        justOnce(Output, thereafter = Terminated) // Message for me
      } {
        mux(inRegion) { Bubble } { External }
      }
  )
}

def chat(centre: ID, newMsgs: Set[Msg]): Iterable[Msg] = {
  val (_, parentToCentre) = gradientWithParent(centre == mid)
  val dependentNodes = rep (Set.empty[ID]) { case (s: Set[ID]) =>
    // nodes whose path towards centre passes through me
    mid + excludingSelf.unionHoodSet[ID](
      mux (nbr{parentToCentre}==mid) { nbr(s) } { Set.empty[ID] }
    )
  }

  statusSpawn[Msg,ChatArgs,Msg](
    process = chatProcessLogic(_), // note: m(_) turns method m to lambda
    newKeys = newMsgs,
    args = ChatArgs(parentToCentre, dependentNodes)
  ).values
}
```

Figure 30: SCAFI code for the opportunistic chat case study.

failures by letting processes run within a wider channel with no coordinator.

The experiment simulates a mesh network of 1000 devices randomly deployed in the urban area of Cesena, in Italy[10]. The experiment, whose software

---

[10]This city has been selected mostly because it is well-known by the authors, and thus inconsistencies in the simulated deployment or bugs related to the mimicking of a real world

part is stable, could be reproduced on different urban settings by changing the deployment area, provided that the target city is reasonably well covered by OpenStreetMap data. A snapshot of a displacement is provided in Figure 29. We simulate the creation and delivery of messages among randomly chosen nodes, with nodes sending a message in time window $[0, 250]$ with an exponential distribution of 1mHz : on average, a message per second is generated by the network. Devices execute rounds asynchronously at an average of 1Hz, though their actual execution frequency can vary according to a Weibull distribution with $k = 0.2$, and such variance varies with the same Weibull distribution, in order to simulate the potential drift of devices executing the program. In each experiment, we generate a different random displacement, different message sources and destinations, and different random seeds for the drift distributions.

We gather a measure of QoS and a measure of resource usage. We use the probability of delivering a message with time as a QoS measure, and we measure the number of payloads sent by each node as a measure of impact on performance. In doing so, we suppose the payload makes up for the largest part of the communication: our metric gets closer to reality as the ratio between the payload size and the total amount of information sent grows. As such, this metric fits particularly well a scenario in which multimedia data are exchanged.

6.1.4. Results and Discussion

Figure 31 shows experimental results of the simulation of the 1000 devices displaced.   The two implementations achieve a very similar QoS, with the flood implementation being faster on average. This is expected, as flooding the whole network also implies sending through the fastest path—by echoing all messages to all neighbours also the shortest path between sender and receiver will be selected. The difference, however, is relatively small, as a message sent using spawn-chat takes few additional seconds to get delivered with devices working at 1Hz on average.

On the contrary, we see the *spawn chat* affords a dramatic decrease in bandwidth usage, by properly constraining the expansion of message delivery bubbles, despite the simplistic selection of the coordinating device: the bandwidth, measured in payloads sent per second per node, is orders of magnitude lower. This is due to the spawn chat message delivery being untied from the number of nodes: a channel gets built from the source to the coordinator and then to the receiver, and such channel length is likely to grow with the network diameter, not with the number of nodes and density of neighbourhoods. On the contrary, the payloads shared by the flood chat are directly tied to the total number of communication links among nodes, as messages get sent to every neighbour on every round.

Those two effects, combined, provide the results of Figure 31, where the spawn chat system trades a moderate degradation of QoS for a considerable improvement over resource usage.

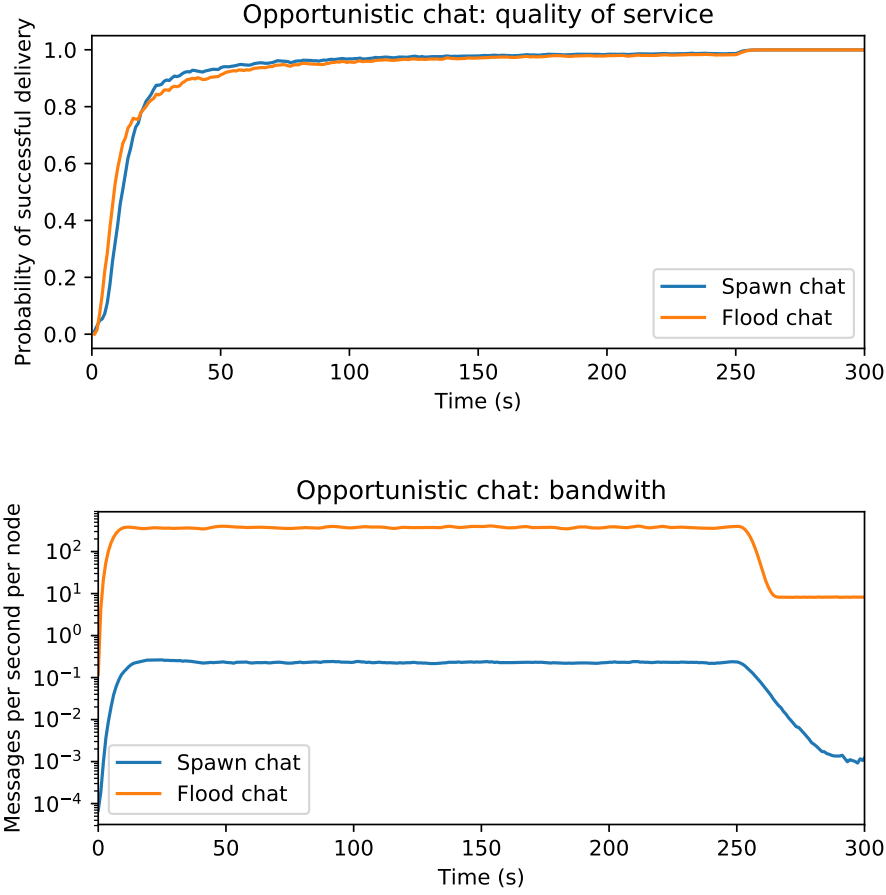displacement could be addressed more easily.

Figure 31: Evaluation of the opportunistic chat algorithms. The figure on top shows similar performance for the two algorithms, with the *flood chat* featuring a slightly better delivery time for the payloads (as it intercepts the optimal path among others). However, as the bottom figure depicts, *spawn chat* requires orders of magnitude less resources due to the algorithm executing on a bounded area (i.e., by involving only a subset of system devices for each message delivery process).

6.2. Reconnaissance with a Drone Swarm

6.2.1. Motivation

Performing reconnaissance of areas with hindrances to access and movement such as forests, steep climbs, or dangerous conditions (e.g. extreme weather and fire) can be a very difficult task for ground-based teams. In those cases, swarms of unmanned airborne vehicles (UAVs) could be deployed to quickly gather information [40]. One scenario in which such systems are particularly interesting is fire monitoring [41].

With this case study, we show how aggregate processes enable easy program-

40

```
def gossipNaive[T](value: T)(implicit ev: Bounded[T]) =
  rep(value)(max => ev.max(value, maxHoodPlus(nbr(ev.max(max, value)))))

def gossipGC[T](value: T)(implicit ev: Bounded[T]) = {
  val leader = S(grain = Double.PositiveInfinity, nbrRange)          // S
  val potential = gradient(leader)                                   // G
  val collectedValue = C[Double,T](potential, acc = ev.max(_,_),     // C
                                   local = value, Null = ev.bottom)
  // Broadcast the "collectedValue" only from where "leader" is true
  valueBroadcast(leader, collectedValue)
}

def gossipReplicated[T:Bounded](value: T, p: Double, k: Int) =
  (replicated{ gossipNaive[T] }(value,p,k) // returns a Map[Long,Double]
   + (Long.MaxValue -> value) // default, lowest-priority map entry
  ).minBy[Long](_._1)._2 // projects the value of instance with min PID
```

Figure 32: Code of the gossip algorithms used in the reconnaissance case study.

ming of a form of gossip that supports a precise and quick, self-healing collective estimation of risk in dynamic scenarios.

6.2.2. Experimental Setup

In this case study, we simulate a swarm of 200 UAVs in charge of monitoring the area of Mount Vesuvius in Italy[11] Our goal is to showcase the application of spawn in a challenging scenario, yet we decided to use reasonably realistic data and parameters where available, in order to work with realistic distances and speeds. UAVs move at an average speed of $130\frac{km}{h}$ following a simple exploration strategy: starting from the base station, they visit a randomly generated sequence of ten waypoints, and once done they come back for refuelling and maintenance. UAVs sense their surroundings once per second and assess the local situation by measuring the risk of fire in a $[0, 1]$ range, where 0 is absence of risk, and 1 is ongoing fire. The goal of the swarm is to agree on the area with the highest risk of fire and report the information back to the station for deployment of ground intervention. A snapshot of the drones performing the reconnaissance is provided in Figure 33.

In this paper, we are not concerned with realistic modelling of fire dynamics, but rather with challenging the distributed algorithms. We designed the risk of fire to be maximum in a random point of the surveyed area for 20 minutes.

---

[11]This area has been heavily hit by wildfires in 2017, exposing volcanic rock to erosion with harmful implications on the local hydrogeological risk (`http://archive.is/j3lsm`). All drones are considered to be dispatched from the same base station in Boscoreale, a town just shy of 30.000 residents located on the southern side of the volcano. The site, situated within the Vesuvio National Park, has been chosen as it matches the location of an actual civil protection headquarter.
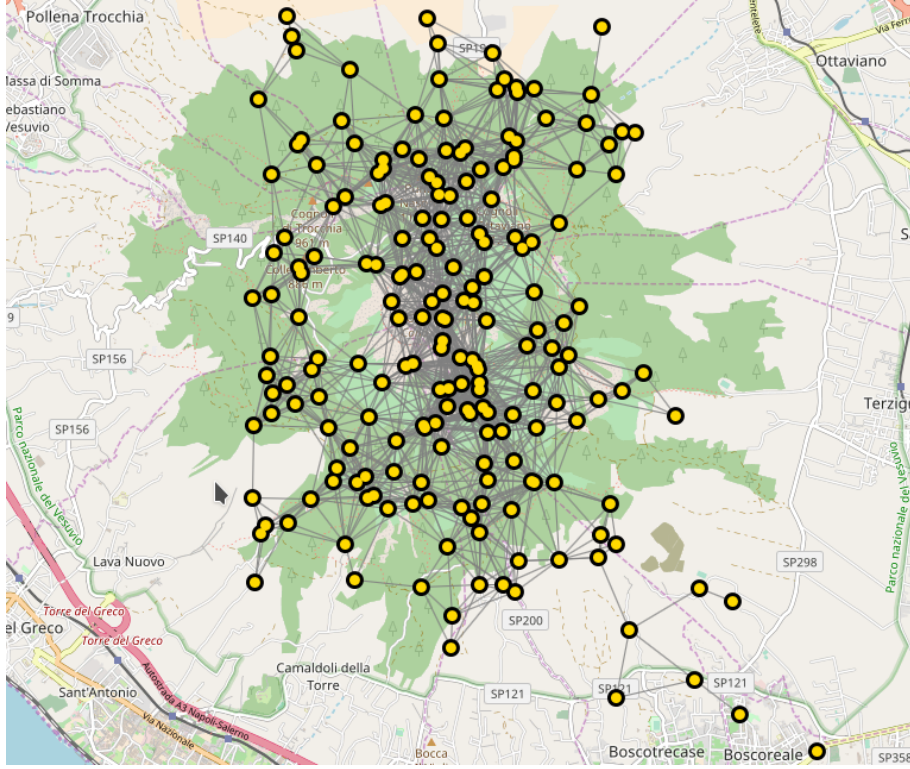
41

Figure 33: Snapshot of the UAV swarm surveying the Vesuvius area as simulated in Alchemist. Yellow dots are UAVs. Grey lines depict direct drone-to-drone communication. Drones travel at an average speed of $130 \frac{km}{h}$, in line with the cruise speed performance of existing military-grade UAVs (see `http://archive.is/8zar5`), and communicate with other drones within 1km distance in line-of-sight. Forming a dynamic mesh network using UAV-to-UAV communication is feasible [42], although challenging [43].

The risk then drops, e.g. due to a successful fire-fighting operation, with the new maximum, lower than the previous, being in another randomly generated coordinate. After further 20 minutes the risk sharply increases again to on a third random coordinate (e.g. due to dry and windy conditions).

We compare three approaches: *(i) naive gossip*, a simple implementation of a gossip protocol estimating the collective maximum by repeated propagation and aggregation of state estimates between neighbouring devices [44]; *(ii) S+C+G* [45], a more elaborated algorithm – based on self-stabilising building blocks [23] – that elects a leader, aggregates the information towards it, then spreads it to the whole network by broadcast; *(iii)* `spawn`-*based replicated gossip*, which replicates the first algorithm over time as per [33] and whose implementation, shown in Figure 32, uses function `replicated` defined in Section 5 upon `spawn`. For the replicated algorithm, we use two replicates ($k = 2$) and choose the period `p` according to the recommendations in [33] by estimating the net-
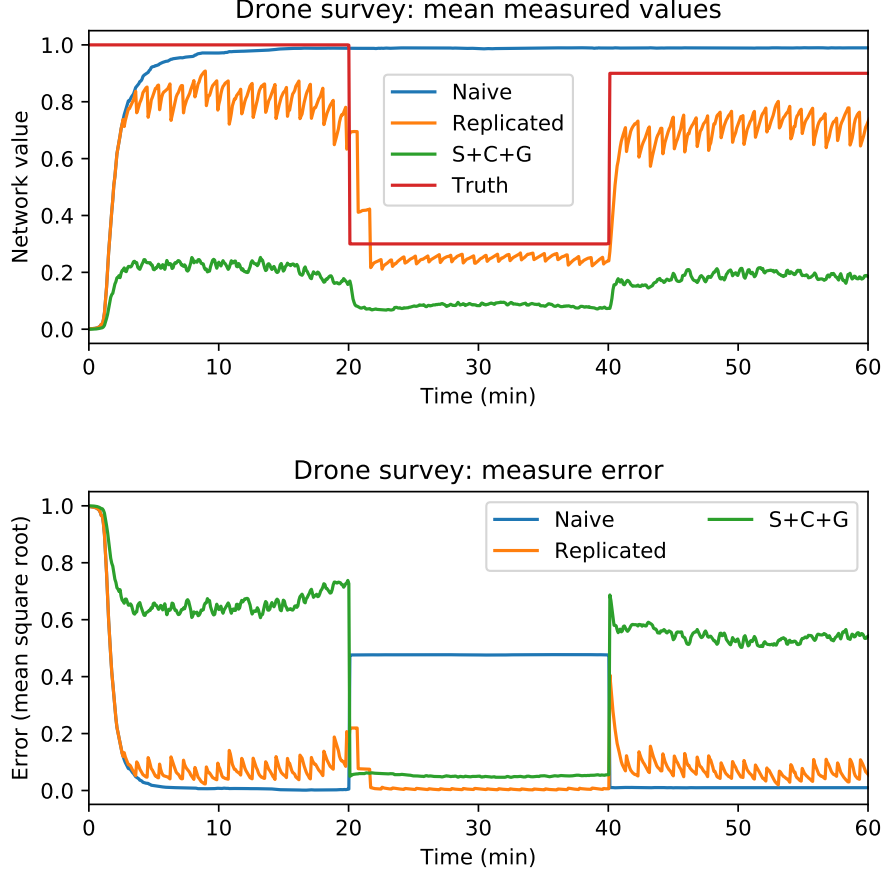
Figure 34: Evaluation of the gossip algorithms in the UAV reconnaissance scenario. The figure on top shows expected values and measures performed by the competing algorithms. The bottom figure measures the error as root-mean-square: $\sqrt{\frac{\sum_n (v_n - a)^2}{n}}$ where n device count, a actual value, and v_n value at the n-th device. The naive gossip cannot cope with danger reduction, S+C+G cannot cope with the volatility of the network, while `spawn`-based replicated gossip provides a good estimate while being to cope with changes.

work diameter offline as the distance between the base station and explorable point farthest from it, divided by the communication range.

6.2.3. Results and Discussion

Results are shown in Figure 34. The naive gossip algorithm quickly converges to the correct value, but then fails at detecting the conclusion of the dangerous situation: it is monotonic in nature [46], hence the collective state converges to the highest peak and can not react to an improvement in the global situation; namely, it does not detect the case in which there is a new global maximum

which is smaller than a previously computed maximum. This is the case after simulation time 20 in Figure 34: changes from the initial situation go undetected, making the naive gossip unsuitable for evolving scenarios. One possible way around would be to regularly restart the protocol, or timestamp readings and discard them after a period of time. Both strategies however incur in large errors upon restart/discard [33], and the former also potentially introduces synchronisation concerns.

S+C+G can adapt to changes, but it is very sensitive to modifications in the network structure: data gets aggregated along a spanning tree generated from the dynamically chosen coordinator, but in a network of fast-moving airborne drones such structure gets continuously disrupted.

In this study, the `spawn`-based replicated gossip performs best, as it conjugates the stability of the naive gossip algorithm with the ability to cope with reductions in the sensed values. The algorithm, in this case, provides underestimates, as it reports the highest peak sensed in the time span of validity of a replicate, and drones rarely explore the exact spot where the problem is located, but rather get in its proximity.

6.3. Self-Discovery of Services in Edge-Cloud Infrastructures

6.3.1. Motivation

As already mentioned, edge computing is an emerging paradigm where storage and computation are brought closer to data sources and users, i.e., towards the "edge of the network". Edge computing is not to be intended as a replacement for traditional cloud computing but rather as a *complementary paradigm*, providing options to system designers when, e.g.: the cloud is temporarily or permanently not available; the cloud is available but undesirable or incompatible with cost, latency or other non-functional requirements; or when the kind of services to be provided operate inherently at the edge, such as in mobile crowdsensing [47] scenarios. In this context, various issues arise including, for instance, QoS-aware deployment of services in heterogeneous infrastructures [48, 49, 50], opportunistic computing [2, 3, 51], adaptive coordination of resources in edge-clouds [52, 53], energy-aware task offloading [54], etc. In this section, we focus on the problem of *decentralised service discovery* [55], which is especially non-trivial in dynamic, open, large-scale edge systems such as, for instance, those envisioned in the *Social* [56] and *Robotic* [57] *IoT*.

Our model of the problem, illustrated in Figure 35, is based on a hierarchical network of nodes organised around three layers: edge, fog, and cloud. Each node can provide and consume multiple *services* such as storage, voice recognition, image processing, etc. In particular, a *task* needs one or more service instances to be performed. Since the services needed to accomplish a task are usually not all available at the node responsible for the task, and the network is potentially dynamic, a service discovery process must be activated to find the set of required service providers. In this case study, we model any service discovery process as an aggregate process spreading in the network. We show how the abstraction can be used to tackle relevant problems in edge computing, and how a process-
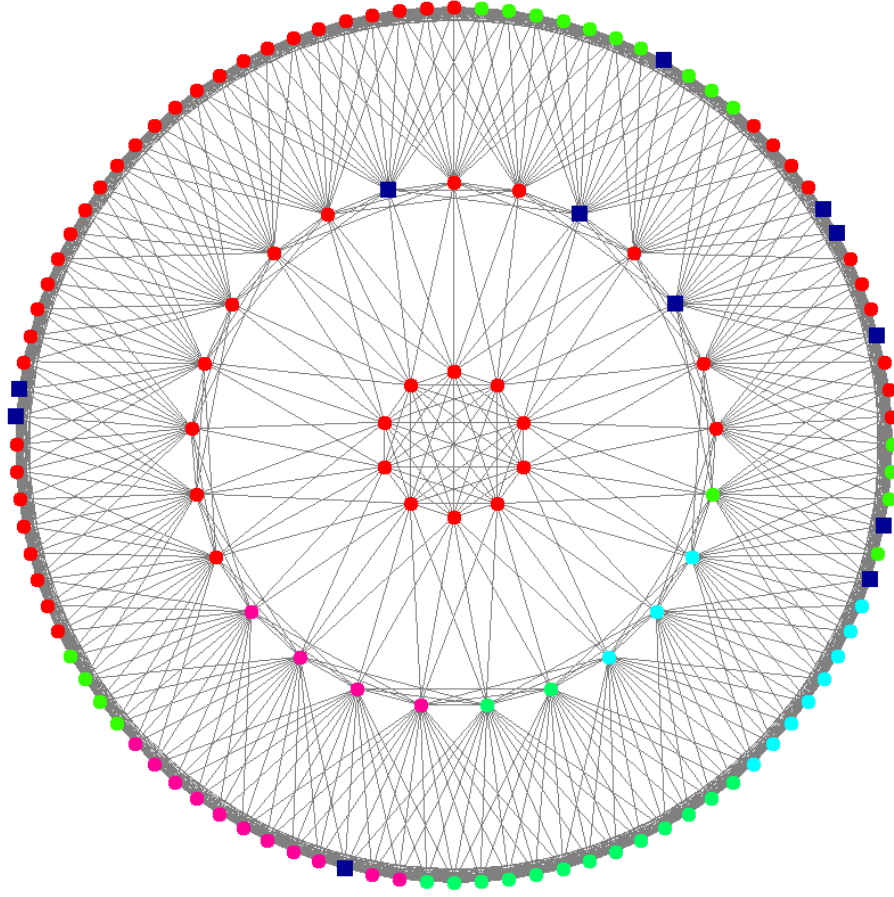
Figure 35: Simulation snapshot of the system for the self-discovery case study, showing the network structure. Service requestors are depicted a dark blue squares, and progressively extend their search for nearby providers. Network nodes potentially working as providers are depicted as circles, matching colours indicate the execution of the same discovery process.

based solution outperforms a corresponding baseline field implementation that does not exploit processes.

6.3.2. Experimental Setup

In this experiment, we model a hierarchical network comprised of diverse systems whose computational performance decreases towards the edge of the network. All network nodes feature both local connection to other nodes (emulating local area networks, or body area networks) as well as Internet access to the cloud. The deployment shape is depicted in Figure 35, and it comprises 100 lightweight devices (with smartphone / system-on-chip level capabilities) on the outer side, 25 devices with desktop PC-level performance, and 10 high performance small servers at the network core. Peripheral nodes run applica-

tions which require one or more remote services. These services can be accessed via the cloud access, or through a nearby edge server hosting them, provided that such node services can be discovered in the local network. Requestors are programmed to try to self-discovery nearby services, and fall back to cloud if none are found shortly after the application launch.

In this experiment, we compare two approaches to self-discovery. The former is a classic aggregate programming solution: requestors diffuse a gradient field, expanding a service request progressively on the edge network. Edge service providers reached by such field can offer their service—a communication channel is established by means of the converge-cast operator C. This network segmentation and communication pattern is also known as self-organizing coordination regions (SCR) [45]. The second approach leverages processes to improve the search capabilities of the baseline. In particular, the main limitation of a plain SCR implementation in the proposed setup is the absence of request overlaps: each request generates an area, and other requests do not penetrate it. On the contrary, aggregate processes allow concurrent requests to overlap in the edge network, enabling a longer range discovery, and potentially more requests to be served without cloud access.

In our experiments, we simulate a request creation process following an exponential distribution with varying λ (request creation rate). In order to emulate the usually sequential use of mobile and web applications, we allow new requests from client devices to be created only if the previous one has been satisfied, either via edge or cloud. Our metric is the number of hops the communication must go through. This metric would basically be the same as running the `traceroute` command from requestor to provider and counting the entries. We believe this roughly maps to latency, as shortest network paths usually count less hops and geographic distance. Since access to a cloud service cost, in terms of hops, varies depending on the backbone network serving the edge deployment, we run simulation with a variable cost of access the cloud. We experimented with a wide range of costs for cloud access, including the unrealistic hypothesis of no cost (which would imply access to the cloud having the same performance of the service being provided by an infinitely powerful device with network performance equal to the loopback network interface). We believe a typical realistic cloud cost can range between 5 and 20 hops.

In our evaluation, we consider a reasonable performance to be up to about 20 hops.

6.3.3. Results and Discussion

Results are depicted in Figure 36. The process-based discovery, leveraging overlapping, is able to timely discover available edge services in face of much higher requests: while the baseline algorithm's performance degrade and cause cloud fallbacks consistently at about 0.00056 requests per user per second, the process-based version performance drop-off point is at 0.0075 requests per user per second, an order of magnitude higher.

Failures in local service discovery and fallback to cloud access are the cause of better performance for the baseline algorithm in the utopian case of no cost
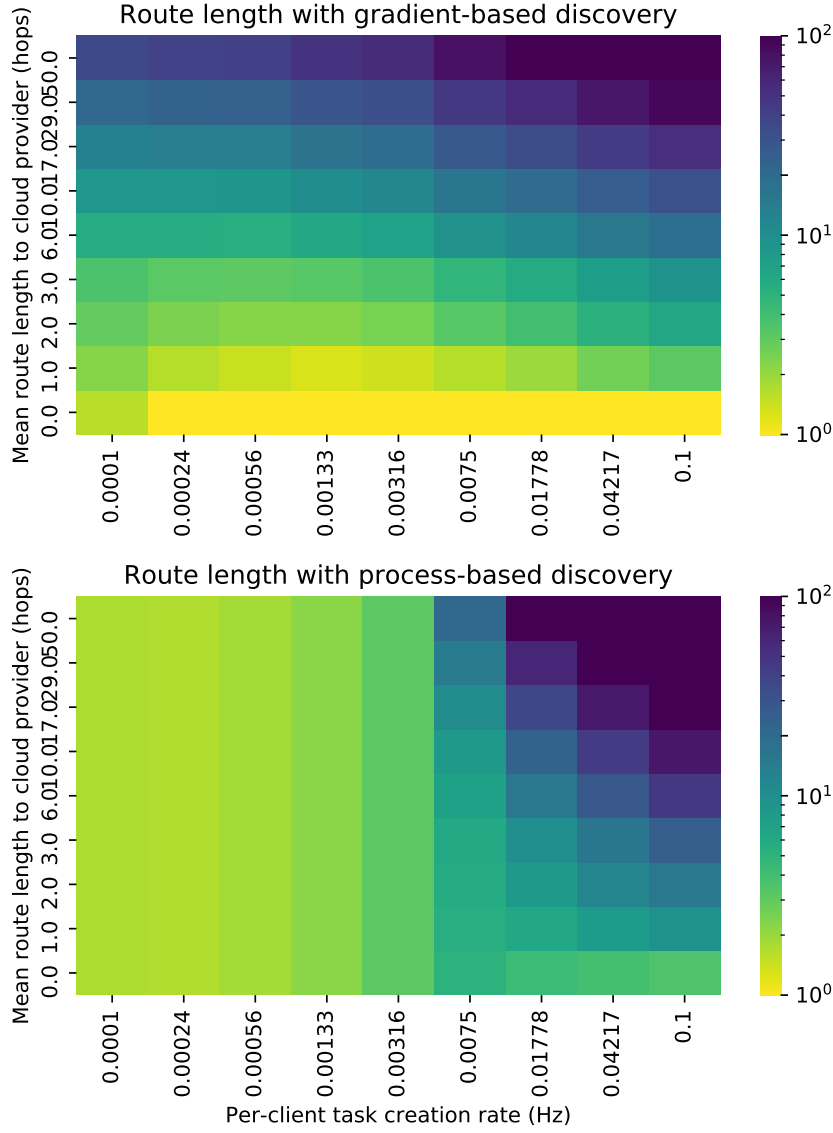
Figure 36: Evaluation of the edge discovery service performance for classic gradient-based SCR implementation (top) and for the overlapping aggregate processes implementation (bottom). Yellow and green colours indicate better performance (fewer hops to reach the service). Purple-ish colours indicate low to unacceptable performance. The process-based approach shows a much higher probability of timely discovering services in the edge network with respect to the baseline algorithm. This fact can be noted by looking at the lower costs across the leftmost columns, up to 0.01778 service requests per second per user, and in particular with realistic cloud access costs.

for cloud access, or in the unlikely cases of access to cloud being cheaper than access to a nearby network node.

7. Related Work

7.1. Spatial Computing

Space is important in computing, both for efficiency and design. The importance of the environment is widely recognised in multi-agent systems design [58]. Indeed, in many cases, systems are *situated* in physical or logical space, and must deal with its constraints and opportunities. In a further view, a situated system can also be thought as the very space it occupies and perceives, so that programming the system is like programming the corresponding space.

Various programming models and approaches that address peculiarities of space-time have been developed across a wide variety of application domains. The survey in [16] describes the historical evolution of "aggregate computing" from research in distributed systems, coordination languages, and spatial computing. In particular, four main clusters of approaches can be identified.

A first cluster is given by "bottom-up" approaches that abstract individual networked devices. In *Tuples On The Air (TOTA)* [59], tuples spatially distributed in space are used to represent context (as information with the environment) and support loosely coupled coordination between situated devices. Aggregate processes can be used to represent situated and evolving tuples à la TOTA [60]. In Hood [61] and Abstract Regions [62], geographic or topological neighbourhoods can be defined to support communication. This logical model aims to simplify application development by abstracting interaction through a region-based collective communication interface. Similarly, the aggregate computing model, upon which aggregate processes are developed, also considers "bulk communications" based on a logical notion of neighbourhood. Moreover, note that two kinds of neighbourhood-like notions exist in aggregate computations: those providing the basic connectivity in the system and "assumed" by the program (i.e., reified by the underlying platform and network) and those at the application-level that are explicitly regulated by branching constructs. In the latter sense, aggregate processes can be used to define dynamic domains constraining the scope of communications.

Another cluster is given by languages for expressing spatial and geometric patterns. Examples include Growing Point Language (GPL) [63], which uses a botanical metaphor with growth processes replicating tropisms in plants, and Origami Shape Language (OSL) [64], which provides geometrical constructs to create and compose regions of a "computational surface". A more recent framework is Pleiades [65], a topology programming framework which exploits self-organising overlays and assembly-based modularity to construct and enforce self-stabilising structural invariants. These features can be useful, e.g., for morphogenesis and self-assembling robots. Aggregate processes can in principle support the development of geometrical shapes, by properly regulating the process of expansion and shrinking of their border together with the mobility of devices. This feature is expected to be relevant to, e.g., modular robotics, but specific investigations are needed to properly validate the effectiveness of the approach in that and similar domains.

Many spatial computing approaches also arose in the context of wireless sensor networks (WSN), where a multitude of sensors are deployed to capture and process environment data [66]. In this context, a common problem is to identify abstractions to simplify streaming and summarising of information over space-time regions. Note that aggregate processes propagating or collecting data through gradients are a very natural way to do that. Similarly, in literature, these tasks are often achieved through *macro-programming* languages that express the system-level behaviour through global abstractions. Notable examples include *Regiment* [67], a functional reactive language which models network state as time-varying signals (similar to computational fields); *SpaceTime Oriented Programming (STOP)* [68], a Ruby DSL that support on-command and on-demand data collection through mobile agents; and *Sense2P* [69], a logic macro-programming language whereby WSNs are abstracted as database upon which data collection and spreading of logic queries can be performed. Aggregate computing is also a macro-programming paradigm, but it is unique as it adopts a functional approach (rather than logic or procedural), which hence grants the benefit of purely declarative compositionality.

Finally, there are general-purpose space-time computing models, such as MGS [70], the field calculus [18], and the Soft Mu-calculus for Computational fields (SMuC) [11]. MGS follows a topological computing approach by which the programmer defines computations over manifolds, whereas the field calculus and SMuC work on a computational field abstraction as covered in this paper.

7.2. Collective Adaptive Systems and Ensemble-based Approaches

Other works that do take into account the collective dimension of systems include so-called ensemble-based approaches, which are centred around the notion of an *ensemble*, i.e., a dynamic formation of components. For instance, in *Distributed Emergent Ensembles of Components (DEECo)* [71], components can communicate by dynamically binding together through ensembles, which are formed according to a *membership* condition and consists of one coordinator and multiple members interacting by *implicit knowledge exchange*. Protelis [38] is an aggregate programming language based on the field calculus, and as such featuring many base constructs already found in SCAFI. It allows arbitrary alignment via the `alignedMap` construct, enabling some form of parallel aggregate execution exploited in the `multiInstance` API [72]. These constructs have inspired `align` and `spawn`, which however (i) have been given precise semantics [21], (ii) are typed, (iii) locally keep the state of active keys from round to round, and (iv) provide automatic propagation and filtering of process keys to the neighbourhood, therefore simplifying domain management.

Another key representative is *Service Component Ensemble Language (SCEL)* [73], a kernel language to specify the behaviour of autonomic components, the logic of ensemble formation, as well interaction through *attribute-based communication* (which enables implicit selection of a group of recipients). A stochastic process algebra with a similar setup is *CARMA (Collective Adaptive Resource-sharing Markovian Agents)* [74]. The idea of these approaches is

49

to use attributes to support dynamic definition of ensembles. An aggregate process also defines and is therefore run by an ensemble: the logic of membership can be purely local or collective, and can unfold progressively in a decentralised fashion; attributes could also be used to regulate the domain of an aggregate process instance or multi-hop multicasting activities.

7.3. Multi-Agent System and Organisations

Multi-agent systems can bring agents together according to multiple organisational paradigms [17], including *(i) network organisations* or *adhocracies*, with complex and dynamic structures; *(ii) hierachies*, with tree-like structures; *(iii) holarchies*, i.e., hierarchically nested structures of *holons* (which are both *wholes* and *parts*) with cross-tree interactions; *(iv) coalitions*, i.e., short-lived, goal-directed groups of agents with the goal of maximising individuals' utilities; *(v) teams*, i.e., sets of cooperative agents which have agreed to work together towards a common goal; *(vi) congregations*, i.e., long-lived agent groupings, formed with no specific goal in mind, aimed at facilitating the process of finding collaborators (cf., service discovery); *(vii) societies*, i.e., long-lived, open organisations aimed at providing consistency through social laws to facilitate coexistence and ordered-yet-flexible interaction; *(viii) federations*, i.e., groups of agents which have ceded some autonomy to a single delegate which represents the group and mediates interaction with other groups; *(ix) markets*, i.e., organisations of competitive buyers, suppliers, and sellers, mainly aimed at supporting processes of allocation and pricing; *(x) matrix organisations*, i.e., structures with rows of agents and columns of managers. Notice that certain kinds of organisations – such as teams, coalitions, congregations, and societies – are structurally similar and rather defined by their dynamics.

With aggregate processes, it is possible to program the logic of group formation and dissolution in order to implement various grouping strategies. In the messaging case study, e.g., a dynamic, goal-directed *team* of devices is formed just to connect senders with recipients, dissolving when the task is completed. The use of aggregate processes to define *social processes* can be considered as an interesting future work.

7.4. Declarative Parallel Programming Models

Declarative programming has the advantage of specifying a desired computation logic without delving into low-level aspects of its actual implementation: in this way, the runtime engine has the flexibility of organising execution while taking into account various contextual aspects and performance metrics. A well-known example is the *query optimiser* component in relational databases, which returns efficient *query execution plans* for a given query expressed declaratively in the SQL language. Similarly, the aggregate programming paradigm, in virtue of its logical, abstract model, delegates to the aggregate computing platform a whole set of issues related to the execution of an aggregate computation over heterogeneous edge-fog-cloud infrastructure, as explored in [22].

Related to the specifics of process execution, there are different models which aims at simplifying programming of multiple computing nodes as well as analysis of resulting programs. For instance, in Valiant's *Bulk Synchronous Parallel (BSP) model* [75], computations are structured as sequences of rounds followed by synchronisation steps; large-scale graph processing frameworks such as Apache Giraph [76] are inspired by BSP. The execution model of aggregate computing, though technically similar (round-based with communications at the end of each round), is motivated by the need of adapting to context change rather than by the need of enabling accurate performance analysis for a variety of architectures. That is, these approaches have different goals and domains: BSP is for parallel computing, and aggregate computing is for collective adaptive systems.

Modern distributed data processing models (e.g., MapReduce [77] and derived ones such as Apache Spark [78]) also abstract away network structure and trade performance for constrained programming schemas. Similarly, aggregate computing programs abstract from execution details and hence are amenable to different deployments and optimisations. However, note that the emphasis in aggregate computations is on the self-adjusting and correct *eventual* result, rather than on incremental but precise and consistent calculations.

By another perspective, works on service computing [79] tailored to dynamic ad-hoc environments [80] are also relevant but usually neglect the collective dimension and rarely consider open-ended situated activities. The service perspective connects also to utility computing and related efforts for abstracting and automatically managing networking and hardware infrastructure [81]—aggregate processes, by admitting diverse computation partitioning schemas [22], foster this vision.

7.5. Process Algebras

A thread of related work is given by *process algebras* or *process calculi*, whose research line was initiated in the 1970s and 1980s with the independent formulation of *Communicating Sequential Processes (CSP)* [82] by Hoare, *Calculus of Communicating Systems (CCS)* [83] by Milner, and the *Algebra of Communicating Processes (ACP)* [84] by Bergstra and Klop. Historical treatments of the development of process algebras can be found, e.g., in [85, 86]. A significant representative of this research line is the π-*calculus* [87], which models concurrent computation as a set of processes that interact by reading from and writing to shared channels. With respect to its predecessors, the π-calculus also attempts to model *mobility*: it does so by supporting dynamic reconfiguration of the system topology by exchanging channel names over channels themselves. In these formal frameworks, important properties to prove include equivalence between two processes (cf., bisimulation), deadlock freedom, liveness etc.

The idea of core languages like π-calculus or the field calculus is generally to identify a minimal set of operators (core language) for capturing relevant modelling aspects, specifying systems in an abstract sense, and then formally analyse system specifications for properties of interest. The main difference between them lies essentially in their abstraction level and goals, which are quite

different. While the goal of the π-calculus is essentially to describe concurrent communicating processes enacting given protocols, the field calculus describes so-called collective adaptive systems [88], realised as continuously-iterated local computation chunks that denote communication implicitly: executed by a collection of interacting devices this results in a globally coherent (i.e., designed) distributed system. Technically, the differences are many: a π-calculus specification composes processes from atomic ones to larger-scale parallel/distributed systems, while in field calculus composition concerns increasingly complex behaviour of the same set of devices; in π-calculus each interaction act results in a message exchange, while in field calculus messages are exchanged continuously and the specification only "fills" the message content; and finally, of a π-calculus specification one typically observes traces of message exchanges, while the field calculus concerns the computational fields that results from a given event structure [18].

More similar to the field calculus are process algebras for collective adaptive systems that provide operators for group communication. For instance, *CARMA* [74] leverages attributes to dynamically aggregate components into ensembles, and components can interact via guarded broadcasts. In *PALOMA (Process Algebra of Located Markovian Agents)* [89], interaction between agents depends on their location and state—through a *perception function* that affects message reception. These approaches, rooted in attribute-based communication, differ from the field calculus, where communication is a broadcast and the group of recipients is captured by some neighbouring relationship. In [90], the π-calculus is extended with a weighted choice operator and channel types for distribution, broadcast, and aggregation, in order to model and analyse self-organisation in WSNs; however, scope seems limited as validation only considers a clustering algorithm. In summary, crucially, with respect to other approaches, the field calculus naturally leads to programming languages (like ScaFi) that are usable in practice and that enable various kinds of applications where collective adaptation makes sense—cf., crowd management [9], edge resource management [52, 91, 45], distributed sensing [20], cooperative problem solving [92], and so on [16].

8. Conclusions and Future Work

In this paper, we have presented and implemented the notion of *aggregate processes* to model dynamic, concurrent collective adaptive behaviours carried out by dynamic formations of devices. In particular, we have designed this abstraction through the `spawn` construct, which extends the practical expressiveness of the field calculus. In a nutshell, this extension can be thought of as adding a support for concurrent field computations, with fine-grained domain management. We have implemented the `spawn` primitive in the ScaFi language, and covered programming techniques for dealing with the definition, generation, shape regulation, and termination of aggregate processes. Finally, we have implemented three case studies, by simulation, showing the correctness and performance of useful programs that would be infeasible (or very hard) to

write in the original framework. These case studies exemplify how a form of swarm or collective intelligence can emerge out of a computational aggregate by having it play an aggregate program which describes how its devices must behave and coordinate by a global perspective. In this view, aggregate processes extend aggregate computing with a new mechanism for structuring and expressing intelligent activity (which includes the ability of a device to evaluate its context to determine whether it should participate in a process or not).

In future work, it would be interesting to consider adoption of aggregate processes to define *social processes* implementing typical organisational paradigms in multi-agent systems, such as those surveyed in Section 7.3. Indeed, as aggregate processes group together sets of agents that need to coordinate or collaborate on common activities, they could be useful to structure or regulate agent societies. Another compelling direction involves investigating the case for a "collective adaptive operating system", where both system processes and application processes launched by users run in the smart cyber-physical ecosystem across the edge-fog-cloud continuum. This direction would attempt to generalise over recent work in the context of smart cities [91, 92]: the idea is that, e.g., a smart city or a long-lived ecosystem made of a network of computing nodes may sustain its operations through several concurrent activities collectively executed by ensembles of smart devices (including smart sensors and actuators); at the same time, citizens may implicitly or explicitly request city services (e.g., crowd-aware navigation, smart lighting) activating new processes (potentially aggregating several devices). In other words, we envision a platform for collective intelligence sustained by a self-organising aggregate of humans and smart devices.

Acknowledgements

References

[1] E. Ahmed, I. Yaqoob, A. Gani, M. Imran, M. Guizani, Internet-of-things-based smart environments: state of the art, taxonomy, and open research challenges, IEEE Wireless Communications 23 (5) (2016) 10–16 (Oct. 2016). doi:10.1109/mwc.2016.7721736.

[2] T. Leppänen, C. Savaglio, G. Fortino, Service modeling for opportunistic edge computing systems with feature engineering, Comput. Commun. 157 (2020) 308–319 (2020).

[3] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio, M. Viroli, Modelling and simulation of opportunistic iot services with aggregate computing, Future Gener. Comput. Syst. 91 (2019) 252–262 (2019).

[4] S. J. Russell, P. Norvig, Artificial Intelligence - A Modern Approach, Third International Edition, Pearson Education, 2010 (2010).

[5] D. Camacho, S. Kim, B. Trawinski (Eds.), New Trends in Computational Collective Intelligence, Vol. 572 of Studies in Computational Intelligence, Springer, 2015 (2015).

[6] J. Kennedy, Swarm intelligence, in: Handbook of Nature-Inspired and Innovative Computing, Springer, 2006, pp. 187–219 (2006).

[7] R. Casadei, M. Viroli, Collective abstractions and platforms for large-scale self-adaptive iot, in: 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W), Trento, Italy, September 3-7, 2018, 2018, pp. 106–111 (2018). `doi:10.1109/FAS-W.2018.00033`.

[8] W. Shi, S. Dustdar, The promise of edge computing, IEEE Computer 49 (5) (2016) 78–81 (2016). `doi:10.1109/MC.2016.145`.

[9] J. Beal, D. Pianini, M. Viroli, Aggregate programming for the Internet of Things, IEEE Computer 48 (9) (2015) 22–30 (2015). `doi:10.1109/MC.2015.261`.

[10] F. Zambonelli, Toward sociotechnical urban superorganisms, IEEE Computer 45 (8) (2012) 76–78 (2012). `doi:10.1109/MC.2012.280`.

[11] A. Lluch-Lafuente, M. Loreti, U. Montanari, Asynchronous distributed execution of fixpoint-based computational fields, Logical Methods in Computer Science 13 (1) (2017). `doi:10.23638/LMCS-13(1:13)2017`.

[12] Y. Hanada, G. Lee, N. Y. Chong, Adaptive flocking of a swarm of robots based on local interactions, in: 2007 IEEE Swarm Intelligence Symposium, SIS 2007, Honolulu, Hawaii, USA, April 1-5, 2007, 2007, pp. 340–347 (2007). `doi:10.1109/SIS.2007.367957`.

[13] G. Valentini, D. Brambilla, H. Hamann, M. Dorigo, Collective perception of environmental features in a robot swarm, in: Lecture Notes in Computer Science, Springer International Publishing, 2016, pp. 65–76 (2016). `doi:10.1007/978-3-319-44427-7_6`.

[14] R. Freeman, P. Yang, K. Lynch, Distributed estimation and control of swarm formation statistics, in: 2006 American Control Conference, IEEE, 2006 (2006). `doi:10.1109/acc.2006.1655446`.

[15] V. Trianni, M. Dorigo, Emergent collective decisions in a swarm of robots, in: 2005 IEEE Swarm Intelligence Symposium, SIS 2005, Pasadena, California, USA, June 8-10, 2005, 2005, pp. 241–248 (2005). `doi:10.1109/SIS.2005.1501628`.

[16] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, D. Pianini, From field-based coordination to aggregate computing, in: 20th Int. Conf. on Coordination Models and Languages, Vol. 10852 of LNCS, Springer, 2018, pp. 252–279 (2018). `doi:10.1007/978-3-319-92408-3\_12`.

[17] B. HORLING, V. LESSER, A survey of multi-agent organizational paradigms, The Knowledge Engineering Review 19 (4) (2004) 281–316 (Dec. 2004). `doi:10.1017/s0269888905000317`.

[18] G. Audrito, M. Viroli, F. Damiani, D. Pianini, J. Beal, A higher-order calculus of computational fields, ACM Trans. Comput. Logic 20 (1) (2019) 5:1–5:55 (Jan. 2019). `doi:10.1145/3285956`.

[19] M. Viroli, R. Casadei, D. Pianini, Simulating large-scale aggregate mass with alchemist and scala, in: Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016, 2016, pp. 1495–1504 (2016). `doi:10.15439/2016F407`.

[20] R. Casadei, M. Viroli, Programming actor-based collective adaptive systems, in: A. Ricci, P. Haller (Eds.), Programming with Actors: State-of-the-Art and Research Perspectives, Vol. 10789 of Lecture Notes in Computer Science, Springer International Publishing, 2018, pp. 94–122 (2018). `doi:10.1007/978-3-030-00302-9_4`.

[21] R. Casadei, M. Viroli, G. Audrito, D. Pianini, F. Damiani, Aggregate processes in field calculus, in: H. R. Nielson, E. Tuosto (Eds.), Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings, Vol. 11533 of Lecture Notes in Computer Science, Springer, 2019, pp. 200–217 (2019). `doi:10.1007/978-3-030-22397-7\_12`.

[22] M. Viroli, R. Casadei, D. Pianini, On execution platforms for large-scale aggregate computing, in: P. Lukowicz, A. Krüger, A. Bulling, Y. Lim, S. N. Patel (Eds.), Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp Adjunct 2016, Heidelberg, Germany, September 12-16, 2016, ACM, 2016, pp. 1321–1326 (2016). `doi:10.1145/2968219.2979129`.

[23] M. Viroli, G. Audrito, J. Beal, F. Damiani, D. Pianini, Engineering resilient collective adaptive systems by self-stabilisation, ACM Transaction on Modelling and Computer Simulation 28 (2) (2018) 16:1–16:28 (Mar. 2018). `doi:10.1145/3177774`.

[24] G. Audrito, R. Casadei, F. Damiani, M. Viroli, Compositional blocks for optimal self-healing gradients, in: 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA,

September 18-22, 2017, 2017, pp. 91–100 (2017). `doi:10.1109/SASO.2017.18`.

[25] F. Lin, R. Keller, The gradient model load balancing method, IEEE Transactions on Software Engineering SE-13 (1) (1987) 32–38 (Jan. 1987). `doi:10.1109/tse.1987.232563`.

[26] G. Audrito, J. Beal, F. Damiani, M. Viroli, Space-time universality of field calculus, in: Lecture Notes in Computer Science, Springer International Publishing, 2018, pp. 1–20 (2018). `doi:10.1007/978-3-319-92408-3_1`.

[27] F. S. de Boer, C. Palamidessi, Embedding as a tool for language comparison, Inf. Comput. 108 (1) (1994) 128–157 (1994). `doi:10.1006/inco.1994.1004`.

[28] A. Brogi, J. Jacquet, On the expressiveness of coordination via shared dataspaces, Sci. Comput. Program. 46 (1-2) (2003) 71–98 (2003). `doi:10.1016/S0167-6423(02)00087-4`.

[29] N. Busi, R. Gorrieri, G. Zavattaro, On the expressiveness of linda coordination primitives, Inf. Comput. 156 (1-2) (2000) 90–121 (2000). `doi:10.1006/inco.1999.2823`.

[30] G. Audrito, F. Damiani, M. Viroli, R. Casadei, Run-time management of computation domains in field calculus, in: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W), Augsburg, Germany, September 12-16, 2016, 2016, pp. 192–197 (2016). `doi:10.1109/FAS-W.2016.50`.

[31] R. Casadei, M. Viroli, Towards aggregate programming in Scala, in: First Workshop on Programming Models and Languages for Distributed Computing, ACM, 2016, p. 5 (2016).

[32] B. C. d. S. Oliveira, A. Moors, M. Odersky, Type classes as objects and implicits, in: W. R. Cook, S. Clarke, M. C. Rinard (Eds.), Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA, ACM, 2010, pp. 341–360 (2010). `doi:10.1145/1869459.1869489`.

[33] D. Pianini, J. Beal, M. Viroli, Improving gossip dynamics through overlapping replicates, in: A. L. Lafuente, J. Proença (Eds.), Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings, Vol. 9686 of Lecture Notes in Computer Science, Springer, 2016, pp. 192–207 (2016). `doi:10.1007/978-3-319-39519-7\_12`.

[34] D. Pianini, S. Montagna, M. Viroli, Chemical-oriented simulation of computational systems with ALCHEMIST, J. Simulation 7 (3) (2013) 202–215 (2013). `doi:10.1057/jos.2012.27`.

[35] S. Hoyer, J. Hamman, xarray: N-D labeled arrays and datasets in Python, Journal of Open Research Software 5 (1) (2017). `doi:10.5334/jors.148`.

[36] J. D. Hunter, Matplotlib: A 2d graphics environment, Computing in Science Engineering 9 (3) (2007) 90–95 (May 2007). `doi:10.1109/MCSE. 2007.55`.

[37] M. A. Gibson, J. Bruck, Efficient exact stochastic simulation of chemical systems with many species and many channels, The Journal of Physical Chemistry A 104 (9) (2000) 1876–1889 (Mar. 2000). `doi:10.1021/ jp993732q`.

[38] D. Pianini, M. Viroli, J. Beal, Protelis: practical aggregate programming, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015, 2015, pp. 1846–1853 (2015). `doi:10.1145/2695664.2695913`.

[39] M. Haklay, P. Weber, Openstreetmap: User-generated street maps, IEEE Pervasive Computing 7 (4) (2008) 12–18 (2008).

[40] J. Beal, K. Usbeck, J. Loyall, M. Rowe, J. Metzler, Adaptive opportunistic airborne sensor sharing, ACM Trans. Auton. Adapt. Syst. 13 (1) (2018) 6:1–6:29 (Apr. 2018). `doi:10.1145/3179994`.

[41] D. W. Casbeer, D. B. Kingston, R. W. Beard, T. W. McLain, Cooperative forest fire surveillance using a team of small unmanned air vehicles, International Journal of Systems Science 37 (6) (2006) 351–360 (may 2006). `doi:10.1080/00207720500438480`.

[42] E. Frew, T. Brown, Airborne communication networks for small unmanned aircraft systems, Proceedings of the IEEE 96 (12) (2008) 2008–2027 (dec 2008). `doi:10.1109/jproc.2008.2006127`.

[43] L. Gupta, R. Jain, G. Vaszkun, Survey of important issues in UAV communication networks, IEEE Communications Surveys & Tutorials 18 (2) (2016) 1123–1152 (2016). `doi:10.1109/comst.2015.2495297`.

[44] D. Shah, Gossip algorithms, Foundations and Trends® in Networking 3 (1) (2009) 1–125 (2009). `doi:10.1561/1300000014`.

[45] D. Pianini, R. Casadei, M. Viroli, A. Natali, Partitioned integration and coordination via the self-organising coordination regions pattern, Future Generation Computer Systems 114 (2021) 44–68 (Jan. 2021). `doi:10. 1016/j.future.2020.07.032`.

[46] K. Birman, The promise, and limitations, of gossip protocols, ACM SIGOPS Oper. Syst. Rev. 41 (5) (2007) 8–13 (2007). `doi:10.1145/1317379.1317382`.

[47] R. Ganti, F. Ye, H. Lei, Mobile crowdsensing: current state and future challenges, IEEE Communications Magazine 49 (11) (2011) 32–39 (Nov. 2011). `doi:10.1109/mcom.2011.6069707`.

[48] M. Villari, M. Fazio, S. Dustdar, O. Rana, R. Ranjan, Osmotic computing: A new paradigm for edge/cloud integration, IEEE Cloud Computing 3 (6) (2016) 76–83 (Nov. 2016). `doi:10.1109/mcc.2016.124`.

[49] M. Chen, W. Li, G. Fortino, Y. Hao, L. Hu, I. Humar, A dynamic service migration mechanism in edge cognitive computing, ACM Trans. Internet Techn. 19 (2) (2019) 30:1–30:15 (2019).

[50] G. Fortino, C. Savaglio, C. E. Palau, J. S. de Puga, M. Ganzha, M. Paprzycki, M. Montesinos, A. Liotta, M. Llop, Towards multi-layer interoperability of heterogeneous iot platforms: The inter-iot approach, in: Integration, interconnection, and interoperability of IoT systems, Springer, 2018, pp. 199–232 (2018).

[51] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio, M. Viroli, A development approach for collective opportunistic edge-of-things services, Information Sciences 498 (2019) 154–169 (Sep. 2019). `doi:10.1016/j.ins.2019.05.058`.

[52] R. Casadei, M. Viroli, Coordinating computation at the edge: a decentralized, self-organizing, spatial approach, in: Fourth International Conference on Fog and Mobile Edge Computing, FMEC 2019, Rome, Italy, June 10-13, 2019, 2019, pp. 60–67 (2019). `doi:10.1109/FMEC.2019.8795355`.

[53] M. G. R. Alam, M. M. Hassan, M. Z. Uddin, A. Almogren, G. Fortino, Autonomic computation offloading in mobile edge for iot applications, Future Gener. Comput. Syst. 90 (2019) 149–157 (2019).

[54] M. Chen, Y. Hao, Task offloading for mobile edge computing in software defined ultra-dense network, IEEE Journal on Selected Areas in Communications 36 (3) (2018) 587–597 (Mar. 2018). `doi:10.1109/jsac.2018.2815360`.

[55] T. X. Tran, A. Hajisami, P. Pandey, D. Pompili, Collaborative mobile edge computing in 5g networks: New paradigms, scenarios, and challenges, IEEE Communications Magazine 55 (4) (2017) 54–61 (Apr. 2017). `doi:10.1109/mcom.2017.1600863`.

[56] F. Cicirelli, A. Guerrieri, G. Spezzano, A. Vinci, O. Briante, A. Iera, G. Ruggeri, Edge computing and social internet of things for large-scale smart environments development, IEEE Internet of Things Journal 5 (4) (2018) 2557–2571 (Aug. 2018). `doi:10.1109/jiot.2017.2775739`.

[57] P. Simoens, M. Dragone, A. Saffiotti, The internet of robotic things, International Journal of Advanced Robotic Systems 15 (1) (2018) 172988141875942 (Jan. 2018). `doi:10.1177/1729881418759424`.

[58] D. Weyns, A. Omicini, J. Odell, Environment as a first class abstraction in multiagent systems, Autonomous Agents and Multi-Agent Systems 14 (1) (2006) 5–30 (Jul. 2006). `doi:10.1007/s10458-006-0012-0`.

[59] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications: The TOTA approach, ACM Trans. on Software Engineering Methodologies 18 (4) (2009) 1–56 (2009). `doi:http://doi.acm.org/10.1145/1538942.1538945`.

[60] R. Casadei, M. Viroli, A. Ricci, Collective adaptive systems as coordination media: The case of tuples in space-time, in: ACSOS Companion Volume, IEEE, 2020, accepted for publication (2020).

[61] K. Whitehouse, C. Sharp, D. E. Culler, E. A. Brewer, Hood: A neighborhood abstraction for sensor networks, in: G. S. Banavar, W. Zwaenepoel, D. Terry, R. Want (Eds.), Proceedings of the Second International Conference on Mobile Systems, Applications, and Services, MobiSys 2004, Hyatt Harborside, Boston, Massachusetts, USA, June 6-9, 2004, ACM / USENIX, 2004 (2004). `doi:10.1145/990064.990079`.

[62] M. Welsh, G. Mainland, Programming sensor networks using abstract regions, in: R. T. Morris, S. Savage (Eds.), 1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings, USENIX, 2004, pp. 29–42 (2004). URL `http://www.usenix.org/events/nsdi04/tech/welsh.html`

[63] D. Coore, Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (1999). URL `http://hdl.handle.net/1721.1/80483`

[64] R. Nagpal, Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2001). URL `http://hdl.handle.net/1721.1/86667`

[65] S. Bouget, Y. Bromberg, A. Luxey, F. Taïani, Pleiades: Distributed structural invariants at scale, in: 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018, IEEE Computer Society, 2018, pp. 542–553 (2018). `doi:10.1109/DSN.2018.00062`.

[66] X. Fu, G. Fortino, P. Pace, G. Aloi, W. Li, Environment-fusion multipath routing protocol for wireless sensor networks, Inf. Fusion 53 (2020) 4–19 (2020).

[67] R. Newton, M. Welsh, Region streams: functional macroprogramming for sensor networks, in: A. Labrinidis, S. Madden (Eds.), Proceedings of the 1st Workshop on Data Management for Sensor Networks, in conjunction with VLDB, DMSN 2004, Toronto, Canada, August 30, 2004, Vol. 72 of ACM International Conference Proceeding Series, ACM, 2004, pp. 78–87 (2004). `doi:10.1145/1052199.1052213`.

[68] H. Wada, P. Boonma, J. Suzuki, A spacetime oriented macroprogramming paradigm for push-pull hybrid sensor networking, in: Proceedings of the 16th International Conference on Computer Communications and Networks, IEEE ICCCN 2007, Turtle Bay Resort, Honolulu, Hawaii, USA, August 13-16, 2007, IEEE, 2007, pp. 868–875 (2007). `doi:10.1109/ICCCN.2007.4317927`.

[69] S. Choochaisri, N. Pornprasitsakul, C. Intanagonwiwat, Logic macroprogramming for wireless sensor networks, International Journal of Distributed Sensor Networks 8 (4) (2012) 171738 (Apr. 2012). `doi:10.1155/2012/171738`.

[70] J.-L. Giavitto, O. Michel, J. Cohen, A. Spicher, Computations in space and space in computations, in: Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 137–152 (2005). `doi:10.1007/11527800_11`.

[71] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, DEECO: an ensemble-based component system, in: P. Kruchten, D. Giannakopoulou, M. Tivoli (Eds.), CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering, part of Comparch '13, Vancouver, BC, Canada, June 17-21, 2013, ACM, 2013, pp. 81–90 (2013). `doi:10.1145/2465449.2465462`.

[72] M. Francia, D. Pianini, J. Beal, M. Viroli, Towards a foundational API for resilient distributed systems design, in: 2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017, IEEE Computer Society, 2017, pp. 27–32 (2017). `doi:10.1109/FAS-W.2017.116`.

[73] R. D. Nicola, D. Latella, A. L. Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, A. Vandin, The SCEL language: Design, implementation, verification, in: Software Engineering for Collective Autonomic Systems, Springer International Publishing, 2015, pp. 3–71 (2015). `doi:10.1007/978-3-319-16310-9_1`.

[74] L. Bortolussi, R. D. Nicola, V. Galpin, S. Gilmore, J. Hillston, D. Latella, M. Loreti, M. Massink, CARMA: Collective adaptive resource-sharing markovian agents, Electronic Proceedings in Theoretical Computer Science 194 (2015) 16–31 (Sep. 2015). `doi:10.4204/eptcs.194.2`.

[75] L. G. Valiant, A bridging model for parallel computation, Communications of the ACM 33 (8) (1990) 103–111 (Aug. 1990). `doi:10.1145/79173.79181`.

[76] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, S. Muthukrishnan, One trillion edges: graph processing at facebook-scale, Proceedings of the VLDB Endowment 8 (12) (2015) 1804–1815 (Aug. 2015). `doi:10.14778/2824032.2824077`.

[77] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113 (Jan. 2008). `doi:10.1145/1327452.1327492`.

[78] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al., Apache spark: A unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65 (Oct. 2016). `doi:10.1145/2934664`.

[79] A. Bouguettaya, M. Singh, M. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, et al., A service computing manifesto: The next 10 years, Commun. ACM 60 (4) (2017) 64–72 (Mar. 2017). `doi:10.1145/2983528`.

[80] C. Groba, S. Clarke, Opportunistic service composition in dynamic ad hoc environments, IEEE Transactions on Services Computing 7 (4) (2014) 642–653 (Oct. 2014). `doi:10.1109/tsc.2013.2295811`.

[81] H.-L. Truong, S. Dustdar, Principles for engineering IoT cloud systems, IEEE Cloud Computing 2 (2) (2015) 68–76 (Mar. 2015). `doi:10.1109/mcc.2015.23`.

[82] C. A. R. Hoare, Communicating Sequential Processes, Springer-Verlag, Berlin, Heidelberg, 2002, p. 413–443 (2002).

[83] R. Milner, Communication and concurrency, Vol. 84, Prentice hall New York etc., 1989 (1989).

[84] J. Bergstra, J. Klop, Process algebra for synchronous communication, Information and Control 60 (1-3) (1984) 109–137 (Jan. 1984). `doi:10.1016/s0019-9958(84)80025-x`.

[85] J. Baeten, A brief history of process algebra, Theoretical Computer Science 335 (2-3) (2005) 131–146 (May 2005). `doi:10.1016/j.tcs.2004.07.036`.

[86] L. Aceto, M. Bravetti, W. Fokkink, A. D. Gordon, Algebraic process calculi: The first twenty five years and beyond, Vol. 75, Elsevier, 2008 (2008). `doi:10.1016/j.jlap.2007.06.001`.

[87] R. Milner, Communicating and Mobile Systems: The π-Calculus, Cambridge University Press, USA, 1999 (1999).

[88] A. Ferscha, Collective adaptive systems, in: Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium

on Wearable Computers, UbiComp/ISWC'15 Adjunct, Association for Computing Machinery, New York, NY, USA, 2015, p. 893–895 (2015). `doi:10.1145/2800835.2809508`.

[89] C. Feng, J. Hillston, PALOMA: A process algebra for located markovian agents, in: Quantitative Evaluation of Systems, Springer International Publishing, 2014, pp. 265–280 (2014). `doi:10.1007/978-3-319-10696-0_22`.

[90] D. Orfanus, T. Heimfarth, F. R. Wagner, Process algebra to model self-organizing behavior in wireless sensor networks, in: Proceedings of the International Conference on Ultra Modern Telecommunications, ICUMT 2009, 12-14 October 2009, St. Petersburg, Russia, IEEE, 2009, pp. 1–6 (2009). `doi:10.1109/ICUMT.2009.5345551`.

[91] R. Casadei, D. Pianini, M. Viroli, A. Natali, Self-organising coordination regions: A pattern for edge computing, in: Lecture Notes in Computer Science, Springer International Publishing, 2019, pp. 182–199 (2019). `doi:10.1007/978-3-030-22397-7_11`.

[92] R. Casadei, C. Tsigkanos, M. Viroli, S. Dustdar, Engineering resilient collaborative edge-enabled iot, in: E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, K. Oyama (Eds.), 2019 IEEE International Conference on Services Computing, SCC 2019, Milan, Italy, July 8-13, 2019, IEEE, 2019, pp. 36–45 (2019). `doi:10.1109/SCC.2019.00019`.

Appendix A. The Scala Programming Language for ScaFi: a Primer

Scala is strongly, statically typed language that coherently integrates the functional and the object-oriented paradigms (with single class inheritance and multiple component composition through traits) and provides advanced typing and composition mechanisms. In the following, we assume familiarity with Java/C-like languages.

Consider the ScaFi `Constructs` trait, reported here for the reader's convenience.

```scala
trait Constructs {
  def rep[A](init: => A)(fun: A => A): A
  def nbr[A](expr: => A): A
  def foldhood[A](init: => A)(acc: (A, A) => A)(expr: => A): A
  def branch[A](cond: => Boolean)(th: => A)(el: => A)
  def mid(): ID
  ...
}
```

In Scala, methods are introduced with the `def` keyword, can be generic (with type parameters specified in square brackets), can be written in curried form (with multiple parameter lists), have a return type which is specified at the

end of the signature, and admit use of both round and curly brackets for 1-ary parameter lists (so that `rep(.)(.)`, `rep{.}{.}`, `rep(.){.}`, `rep{.}(.)` are all valid calls). When using *block syntax* {. . .} – e.g., as a parameter value or body of a method, in `if-else` expressions etc. – it is possible to specify multiple expressions or statements (which are also expressions, returning a value of the singleton type `Unit`); the last evaluated expression is the return value of the block.

```
nbr {
  val x = 10
  x + 2
} // the nbr block is a by-name parameter; when evaluates, it returns 12
```

Moreover, at the invocation side, named arguments can be used:

```
nbr(expr = 1+2)
```

Parameterless methods can also be defined and can be simply invoked without the common function call operator `()`:

```
mid // invokes parameterless method
```

Syntax $\Rightarrow A$ denotes a *call-by-name* parameter which is passed unevaluated—i.e., as a *thunk*—to the method (basically, it is syntactic sugar over nullary functions). E.g., expression `1+2` in the above `nbr` call is not evaluated strictly but wrapped as such and evaluated any time argument `expr` is referred to within `nbr`'s implementation.

Syntax $(T_1, \ldots, T_n)$ denotes tuple types, whose objects $(v_1, \ldots, v_n)$ can be queried for elements via selectors $t.\_n$. Syntax $(k \rightarrow v)$, useful for maps, can be used to denote a two-element tuple (k, v):

```
val m = Map("a"->1) + ("b"->2) // m = Map("a"->1,"b"->2)
// Note '+' overloaded to add entry to map
m.toSet // (Set("a",1),("b",2)) -- i.e., a map is a set of 2-elem tuples
```

Keyword `val` introduces named immutable references to (mutable or immutable) objects. Notice that we omitted the type of `m`, by exploiting Scala's type inference.

Syntax $A_1 \Rightarrow \ldots A_n \Rightarrow Ret$ denotes (curried) function types, whose values (functions) can be expressed with similar syntax $(a_1 : A_1) \Rightarrow \ldots \Rightarrow (a_n : A_n) => retExpr$. A shortcut for creating lambdas (anonymous functions) leverage symbol _ for subsequent positional arguments.

```
((x:Int)=>x+1)(2) // => 3
((_:Int)+1)(2)    // => 3
(_+1)(2)          // => 3 (with type inference)
```

Functions can also be obtained from methods (methods can be defined in class, trait or object definitions, and even locally to other methods):

```
def m(x: Int)(y: Int): Int = ???
val mfun = m _  // Or also: m(_). Inferred type: Int => (Int => Int)
```

Symbol `???` is a valid expression, of type `Nothing` (the bottom type in Scala, i.e., the subtype of any other type), that stands for a missing implementation (i.e., it raises an exception when evaluated). Lambdas can also be defined through pattern matching on the input parameter,

```
{
  case pattern1 => body1
  //  ...
  case patternN => bodyN
}
```

These can be used in place of $arg \Rightarrow ret$ syntax to improve code legibility by, e.g., destructuring tuples into values:

```
rep((0,""))(tp => (tp._1+1, tp._2+"!")) // or also: rep(...){ tp => ... }
rep((0,"")){ case (k,s) => (k+1, s+"!") }
```

Pattern matching on values leverage keyword `match`, and may optionally specify guards introduced with keyword `if`:

```
value match {
  case pattern1 if guard1 => body1
  case _ => bodyCatchAll // catch-all pattern
}
```

In Scala, it is possible to express type class constraints on type parameters, leveraging implicit parameters (or an equivalent special syntax). For instance, in

```
def f[T](implicit evidence: C[T]) = ???
def g[T:C] = { // constraint T:C ensures there's a C[T] implicit
  val evidence = implicitly[C[T]] // takes the implicit
  ???
} // exactly like f
```

methods `f` and `g` can only be invoked if there exists in scope an implicit instance of type `C[T]` (defined with keyword `implicit` and possibly `import`ed in scope). This is the idiomatic way to implement the typeclass pattern in Scala [32].

Another useful construct is `case class`es (product types),

```
case class C1(v1: Int)
val c = C1(7) // creates an instance (no need to use the 'new' operator)
c match { case C1(x) => x } // returns 7 (extract by pattern matching)
c == C1(7) // true (field-by-field equivalence)

case class C2[T](x: T, y: T)(val z: T) // equivalence only on x, y
val c2 = C2(0,0)(8) // type inference: C2[Int]
c2 == C2(0,0)(7) // true
```

```
c2 match { case c@C2(a,b) => a+b+c.z } // 8 (c binds to entire structure)
```

which simplify the definition of immutable records and provide built-in support for structural equivalence, pattern matching, and destructuring. Notice that, according to the semantics of Scala's case classes, `C2(0,0)(7)==C2(0,0)(8)` is `true` because equality is based only on the fields provided in the *first* parameter list (which are the same for the left-hand and right-hand side expressions).

Sum types can be defined by using inheritance.

```
trait Status
// A Status can be a BubbleStatus or an ExternalStatus
case object BubbleStatus extends Status
case object ExternalStatus extends Status
```

where we may use `case object`s instead of `case class`es if the component types are singleton (so we need just one instance). Sum and product types are the Scala's idiomatic way of defining Algebraic Data Types (ADTs).

Sometimes, it is handy to define type aliases, for better naming or for defining shorthands.

```
type Pid = Long
```