Future Generation Computer Systems 114 (2021) 44–68

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Partitioned integration and coordination via the self-organising coordination regions pattern

Danilo Pianini *, Roberto Casadei *, Mirko Viroli, Antonio Natali

Department of Computer Science and Engineering, Alma Mater Studiorum—Università di Bologna, Cesena, Italy

ARTICLE INFO

Article history:
Received 4 February 2020
Received in revised form 30 May 2020
Accepted 13 July 2020
Available online 24 July 2020

Keywords:
Coordination
Distributed systems
Design patterns
Self-organisation
Self-improving integration
Edge computing
Aggregate programming

ABSTRACT

In software engineering, knowledge about recurrent problems, along with blueprints of associated solutions for diverse design contexts, are often captured in so-called *design patterns*. Identifying design patterns is particularly valuable for novel and still largely unexplored application contexts such as the Internet of Things, Cyber–Physical Systems, and Edge Computing, as it would help keeping a balanced trade-off between generality and applicability, guiding the mainstream development of language mechanisms, algorithms, architectures, and supporting platforms. Based on recurrence of related solutions found in the literature, in this work we present a design pattern for self-adaptive systems, named *Self-organising Coordination Regions* (SCR): its goal is to organise a process of interconnecting devices into teams, to solve local tasks in cooperation. Specifically, it is a decentralised coordination pattern for partitioned integration and coordination of devices, which relies on continuous adaptivity to context change to provide resilient distributed decision-making in large-scale situated systems. It leverages the divide-and-conquer principle, partitioning (in a self-organising fashion) the network of devices into regions, where internal coordination activities are regulated via feedback/control flows among leaders and subordinate nodes. We present the pattern, provide a template implementation in the Aggregate Computing framework, and evaluate it through simulation of two case studies in edge computing and hierarchical, heterogeneous networks.

© 2020 Elsevier B.V. All rights reserved.

1. Introduction

Emerging distributed computing scenarios, such as those envisioned in the context of the Internet of Things (IoT), Cyber–Physical Systems (CPSs), and Edge Computing, are generally characterised by large scale, heterogeneity, dynamicity, and openness. These aspects, taken together, lead to two main problems. The first is a problem of *integration*: putting and re-arranging devices together into a cohesive structure and network of interactions to effectively and efficiently promote some system function. The second is an *operational problem*: harnessing complexity so as to be able to evolve systems by reacting to change in a timely and cost-effective manner. Given the physical, logical, and temporal scales involved, addressing the above problems through static systems specifications and human-in-the-loop approaches appears unfeasible.

In the vision of *autonomic computing* [1], this issue is to be addressed by making a system itself responsible for interconnecting its own components (and among the others, incorporating addition of new devices), and for making their cooperation fully-adaptive to continuous change (changes of context, faults, and other inherent sources of dynamicity) following the high-level goals expressed by the designers. Specifically, the case when autonomic behaviour is engineered to achieve proper integration of system components, is referred to as the *self-integrating* property, or putting it equivalently, as the problem of *self-adaptive integration*: the system should be instrumented by an autonomous process that continuously creates or adjusts interactional structures and responsibilities in order to promote the pursuing of system goals opportunistically from the set of available devices or components. The significance of the problem is witnessed by the *Self-Improving System Integration (SISSY) initiative* [2], which promotes the study of self-* approaches for *systems of systems* [3], *interwoven systems* [4], and *organic computing* systems [5], where heterogeneity, real-time demands, mutual influences, and unpredictability take over.

The problem of (self-)integration is especially relevant in emergent IoT-based *heterogeneous ecosystems* and *collaborative*

* Corresponding authors.
E-mail addresses: danilo.pianini@unibo.it (D. Pianini), roby.casadei@unibo.it (R. Casadei), mirko.viroli@unibo.it (M. Viroli), antonio.natali@unibo.it (A. Natali).

https://doi.org/10.1016/j.future.2020.07.032
0167-739X/© 2020 Elsevier B.V. All rights reserved.

scenarios where networked ensembles of devices need to operate in their locality by coordinating activities and decision making because, e.g., the information, rights, or resources available at an individual device are not sufficient for it to autonomously carry out the tasks at hand. There, indeed, structures should be continuously adjusted to foster synergistic interactions supporting both individual and collective activity. As fully peer-to-peer approaches tend to be globally inefficient, the *pattern* that emerges in the literature [6,7] for tackling this problem – especially for networks that are large-scale, heterogeneous, and often also open and situated – typically follows the *divide-and-conquer* principle: breaking the system into a number of space-driven subsystems managed by *leader devices* that collect feedback from (and enact decisions to) a subset of other participants (*workers* or *users*). We call this pattern ***Self-organising Coordination Regions (SCR)***, since it works through an internally-regulated, adaptive organisation of devices in regions where activity is coordinated via feedback/control flows among master and worker nodes. Examples of pattern usage, which provide motivation for the current work (and are reviewed in this paper), include self-adaptive software architectures [6,8], decentralised orchestration systems [9,10], crowd management [11,12], robot swarm control [13], wireless sensor networks [14,15], and target tracking [16,17]. We define the pattern by analysing and abstracting from the large amount of corresponding related work, while at the same time exploring and documenting its variety of forms.

This paper provides the following contributions:

1. it presents and discusses a design pattern for partitioned integration and coordination that aims at fostering adaptive, resilient self-integration in large-scale situated systems;
2. it proposes a possible decentralised, self-organising implementation of the pattern in the *aggregate computing* framework [11] that sums to a composition of continuously self-adaptive collective behaviours; and
3. it shows the significance and applicability of the pattern through case studies in the context of layered network infrastructures and *edge computing*.

Accordingly, this work can be considered as a contribution to the SISSY initiative and its goals, addressing issues raised in the areas of *design and architecture* (the pattern), *cooperation and self-* mechanisms* (aggregate computing), and *applications* (edge computing and resource management in hierarchical networks).

This article is an extended version of the conference paper [7], providing new material which can be summarised in the following additional contributions: (i) extensive literature research revealing more known uses of the pattern as well as more related patterns; (ii) contextualisation of the work in the context of SISSY; (iii) improved and extended description of the pattern, including a more complete treatment of its underlying mechanisms and extensions; and (iv) an entirely new case study, demonstrating the usefulness of the pattern in (non-spatially situated) hierarchical networks.

The organisation of the manuscript is as follows. Section 2 presents the target context and provides motivation for the pattern by discussing its sub-patterns, related patterns, and known uses. Section 3 describes the SCR pattern by covering and detailing its goals, structural and dynamical dimensions, applicability issues, variants, and terminology. Section 4 discusses implementation issues and provides an implementation template of the pattern in the aggregate programming framework as a functional composition of collective adaptive behaviours. Section 5 provides empirical evaluation on two diverse scenarios making for use cases of the pattern. Finally, Section 6 draws conclusions and points out directions for future work.

2. Motivation and related work

In this section, we draw motivation for the proposed pattern, in terms of (i) a class of problems in a specific context, (ii) the role of patterns in engineering complex systems, and (iii) related work, including sub-patterns, related patterns, and known uses. We first note there is a significant problem, across various domains, of partitioned orchestration and self-integration in large-scale distributed systems (Section 2.1). Though similar problems may have recurrent solutions in literature, it is only when those are recognised as a pattern that a general and reusable understanding is achieved: we hence discuss sources and implications of design patterns for self-* systems (Section 2.2). We then briefly review the proposed SCR pattern, and contextually cover its sub-patterns (Section 2.3), before comparing it to related patterns (Section 2.4). Finally, we provide evidence that SCR is indeed a *pattern* (in the sense of, e.g., [18]), due to multiple previous independent occurrences in disparate domains (Section 2.5).

2.1. Context

We consider the context of coordination and self-integration in large-scale distributed systems. Specifically, we focus on scenarios – e.g., pervasive computing, Collective Adaptive Systems (CAS), IoT, CPSs, and edge computing – characterised by:

- *Distribution.* Typical assumptions made in concentrated systems cease to hold: indeed, distribution of components inherently leads to concurrency, lack of global time, independent failure, and availability issues [19]—with corresponding consequences.
- *Situatedness.* Components are immersed into an environment, and their operations (activity and interaction) are based on a local context that typically represents a small subset of the entire environment. When such an environment is physical, contexts often reflect spatiotemporal constraints of device location (e.g., inputs and outputs may be limited to physical proximity), but this peculiarity could fade with partially or completely logical environments (e.g., with cloud-based coordination).
- *Heterogeneity.* Components may differ by their nature (e.g., virtual or physical), structure, computational power, energy requirements, controllability, level of autonomy, ability and technology to interact with others, and general dependability.
- *Large scale.* Systems may be too large to be manually operated or orchestrated by a centralised entity. In particular, this does not only affect the technological and pragmatical level of engineering, but also at the logical and design level, where it becomes important to consider abstractions and architectural guidelines to intrinsically and smoothly address the behaviour and quality attributes of systems in a scalable fashion.

More specifically, in such scenarios, a number of *problem forces* must be dealt with. *Heterogeneity* creates asymmetry in individual capabilities, such that *collaboration* is essential to solve complex tasks, e.g., when the information, rights, or resources available at an individual device are not sufficient for it to autonomously carry out the whole task at hand. A *locality principle* holds due to *situation and context* being key for both individual and collective activity: hence, the difficulty (and cost) of interaction (integration, collaboration) typically increases with the (physical or logical) distance between sources, processes, and users. Also, the environment and system structure tend to be highly *dynamic* (e.g., due to uncontrollable processes, mobility, or failure), and to promote *integration* of new components

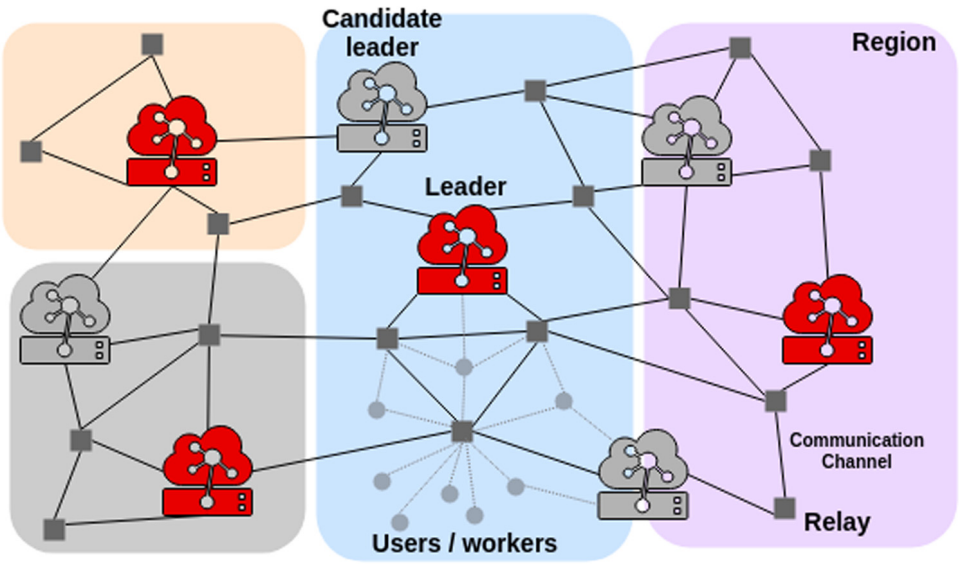


Fig. 1. SCR from a structural perspective—see description in Section 3.4. Among the set of candidate leaders ("gateway-like" grey nodes), one leader ("gateway-like" red nodes) is elected per region (coloured area); intermediary nodes (grey squares) are not eligible for leadership and mediate interaction between leaders and members (grey circles). Edges denote connectivity links. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

(devices, functions) on the fly, creating a situation of constant change where the system stability is continuously endangered by perturbations. The unpredictability of the environment, together with the locality of observations and the complexity (in terms of inter-dependencies) of the phenomena to monitor and control (cf. emergence), fuel the *uncertainty* in observations and planned courses of individual action. Finally, often neither *full centralisation* nor *full decentralisation* in control and decision making is possible or desirable—the former due to the introduction of a single point of failure and reduced scalability, the latter for the inherent complexity in achieving consensus and globally optimised functions.

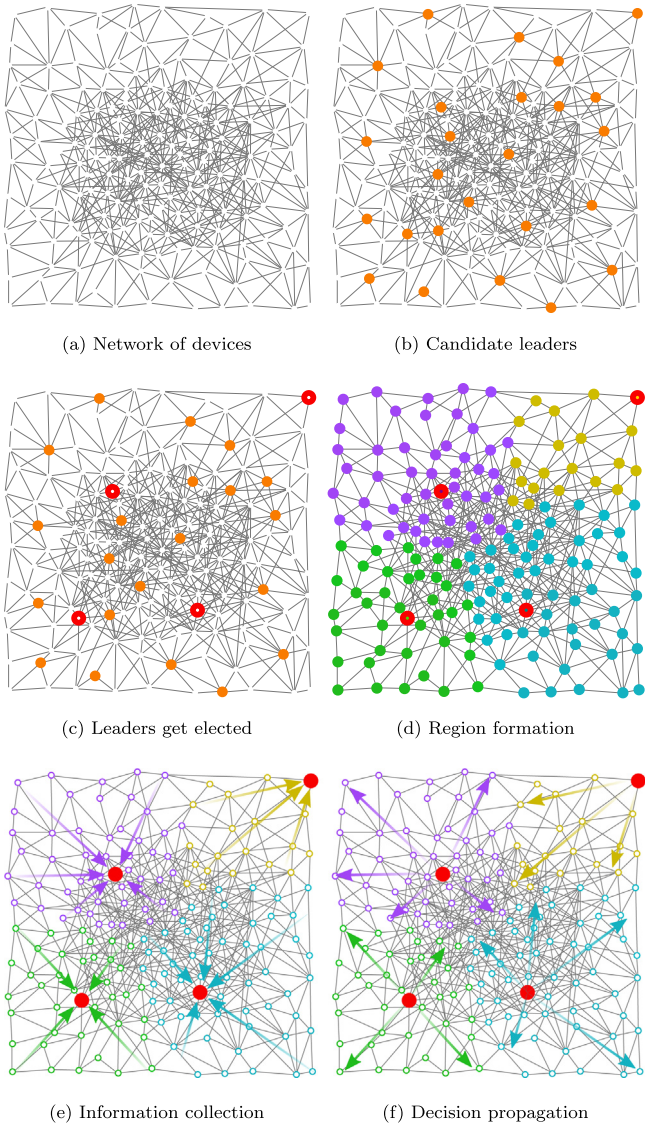
These problems are parts of the challenges identified in the context of the SISSY initiative [2], which seeks to support solutions to effective and efficient cooperation among distributed, heterogeneous components. For instance, the key role of coordination for integrating autonomous systems is recently analysed in [20], highlighting as main research challenges the degree of centrality as well as the impact, overhead, and robustness of coordination. The problem of self-integration characterised in [21], does include that of flexible management of interaction between heterogeneous components in large-scale distributed configurations, and is exacerbated by the increasing interwoven structure [4] and organic nature [22] of such systems.

2.2. Need for design patterns for self-* systems

Design patterns capture expert knowledge by describing reasoned solution blueprints for well-defined classes of repeatedly occurring problems in specific contexts [23] They are thus paramount in software engineering, as they help harnessing complexity by characterising systems of forces arising in a context, and strategies to resolve them [24]. Moreover, they introduce a common vocabulary, fostering the team communication, and rely on it to denote intents and properties of solutions, abstracting from implementation details, and providing motivated guidance towards desired configurations [23]. In the last decades, several classes of patterns have been discovered to promote effective design and implementation of software-based systems, collected in *pattern catalogues* distilling experience, e.g., regarding object-oriented software [18], elemental programming blocks [25], programming language implementation [26], concurrent software design [27], enterprise integration [28], message-based systems [29,30], programming with the event-loop [31], fault-tolerance [32] etc. Additionally, patterns can be organised into taxonomies according to various dimensions (e.g., by abstraction level into architectural, design patterns, and programming idioms [23]), can be considered in relation with other patterns (e.g., through the relationships of combination, variance, and refinement [23]) and can be documented following multiple formats (e.g., the well-known *Alexandrian* [24], *POSA* [23], and *GoF* [18] formats).

In novel and still largely unexplored application contexts; such as the emerging distributed computing scenarios envisioned by pervasive computing, IoT, CPSs, and edge computing; identifying recurrent patterns can be extremely valuable to mainstream development of language mechanisms, algorithms, architectures and supporting platforms—keeping a balanced trade-off between generality, applicability, and guidance. The various challenges emerging in these contexts are often addressed in related research fields, where particular approaches and paradigms are investigated, studied, and proposed, and for which collections of patterns have sometimes been uncovered. Relevant examples of these include early catalogues of multi-agent system architectures [33], multi-agent organisational paradigms [34], biologically-inspired coordination [35,36], collective adaptive behaviour [37], decentralised self-adaptation [8] and self-adaptive coordination [38]. These patterns are usually presented at various abstraction levels, ranging from principles to high-level components and formally defined rules of evolution, and do not typically provide complete, general solutions to the problem of complex adaptive behaviour in large-scale situated systems.

Importantly for the scope of this paper, the SISSY initiative recognises the importance of architectural and design concepts to deal with mutual influence between components, detect and represent emergent phenomena, and support continuous self-reflection [2]. A set of architectural integration patterns in the context of autonomic computing was presented in [39]. They support the resolution of integration-related conflicts within autonomic systems (internally) as well as *systems* of autonomic systems (externally).



(a) Network of devices (b) Candidate leaders

(c) Leaders get elected (d) Region formation

(e) Information collection (f) Decision propagation

Fig. 2. Series of snapshots showing the phases of the pattern.

2.3. Partitioned integration and SCR as a pattern

In order to follow this section, we provide a summary of the pattern (also, refer to Figs. 1–3), which is described in more detail in Section 3:

> The SCR pattern organises the structure and dynamics of the system, which is assumed to be a network of nodes with neighbouring connections, as follows: a system-wide leader election process determines a set of *leaders* among a set of *leader candidates*; the system (or its environment) forms a self-adjusting set of *regions*, each one regulated by a single leader; within each region, a feedback loop is established, whereby the leader receives upstream information flows from the members of the region (possibly leveraging *intermediaries*) and emits a control information flow downstream.

The SCR pattern addresses an important problem of self-* integration: the ability to create in a self-managed way dynamic (and continuously changing) teams of devices to solve in cooperation problems related to a given system locality (or region). The task to solve is then carried on in each region through feedback-and-control loops, between a leader and the other agents/devices in the region. This approach recurs in a number of scientific works and proposed solutions, and is implemented variously. Some patterns presented in the aforementioned catalogues [35, 37] constitute the foundations of the current work. Indeed, the SCR pattern is a combination of three fundamental coordination (sub-)patterns:

- *Multi-leader election*. In distributed systems, it is sometimes useful to break symmetry, sparsely selecting some local centralisation points. This pattern (also known as sparse choice) consists in the election of multiple leaders to uniformly cover a logical or physical space.
- *Information propagation*. Communication patterns that abstract from implementation details or networking protocols are essential in distributed systems. This pattern consists of propagating information from one or more sources outward, independently of the underlying network structure.
- *Information collection*. This pattern consists of collecting information from a set of sources into one sink, still abstracting from low-level details and networking protocols.

Situations where devices can fail or change are accounted for, coherently to the self-organisation principle, by considering the aforementioned patterns as *continuous processes*, or at least as processes *reactive* [40] to failure or change. The continual nature of the involved processes leads to information (updates) that move continuously, as a logical stream (logically, net of potential optimisations), as captured by the *information flow* abstraction [41] as follows:

> *An information flow is a stream of information from source localities towards destination localities and this stream is maintained and regularly updated to reflect changes in the system. Between sources and destinations, a flow can pass other localities where new information can be aggregated and combined into the information flow.*

Information flows are commonly implemented through processes in charge of building and maintaining the communication paths. One notable example of such processes, found in several works in the literature, is the *gradient* [38,42–44], a self-healing [45] distributed data structure mapping any node of the system to its estimated distance from "source" points: it provides a backhaul for controlling the directions of propagation and collection data flows. Information flows can be naturally expressed in the library introduced in [37], which fosters the definition of the collective behaviour of an ensemble of devices through a composition of self-organising and self-stabilising coordination patterns, drawing inspiration from biology [35]. The aforementioned sub-patterns are in fact considered "building blocks" in [11], where they are respectively called (using terse names derived from the similar idea of S, K, I combinators in λ-calculus) *S* (for ***S**parse-choice*—i.e., a scattered selection from the set of participating devices), *G* (for ***G**radient-cast*—i.e., a multicast diffusing information along a gradient), and *C* (for ***C**onverge-cast*—i.e., a multicast aggregating information to a sink device).

2.4. Related patterns

A presentation of architectural patterns for integration in autonomic computing systems can be found in [39]. These patterns support the resolution of conflicts arising when integrating autonomic components and systems together. In SCR, conflicts are avoided *by design* within regions through the corresponding leader nodes (similarly to the *Controller* and *Hierarch* patterns). The election of leaders and formation of regions, instead, involve decentralised conflict resolution (similarly to the *Collaboration*

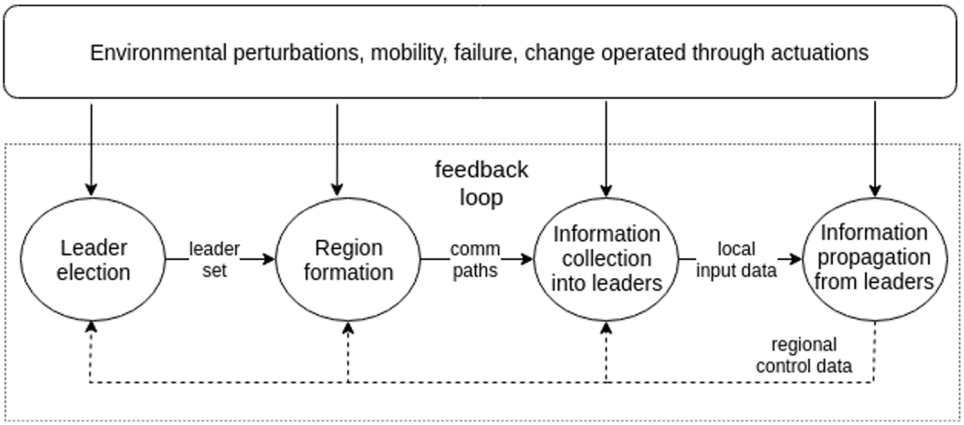


Fig. 3. SCR from a dynamical perspective—see description in Section 3.5. Notation: solid arrows represent required inputs or unavoidable perturbations; dashed lines denote possible feedback loops.

pattern). Moreover, the SCR pattern provides a structure fostering *meta-management* (i.e., for adjusting the leaders or regions configuration).

Very related to SCR is the *Leader–Followers* pattern applied, for instance, in formation control [46], consensus [47], and tracking [48]; not to be confused with the homonym pattern [27] originally proposed in the context of concurrency to efficiently organise a system of threads waiting for events on a shared set of handles. In Leader–Followers, control in distributed and multi-agent system is organised around *leader* agents which take the decisions for a group of agents known as its *followers*. What SCR adds is an organisation in terms of *multiple* leaders, dynamically elected to "effectively" cover a region of the (possibly logical) space.

So, in SCR there is a notion of *spatial region*. By considering a logical space, the region could also be thought of as a way to group agents in order to make them work efficiently on the distributed problem at hand. This is related to *organisational patterns* in multi-agent systems [34], which include, for instance: *(i) network organisations* or *adhocracies*, with complex and dynamic structures; *(ii) hierarchies*, 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; and *(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. In particular, SCR – which could also be seen as a meta organisational pattern – provides a way for expressing dynamic federations— upon which a continuous, decentralised, leader-regulated process is established for coordinating local activity towards increasingly non-local benefits. Beside federations, other organisational paradigms provide for specific aspects (e.g., stability of the regions, or coordination constraints) or variants of the pattern (as covered in Section 3.9).

A well-known organisational meta-pattern for self-adaptive systems, envisioned in the context of autonomic computing, is MAPE [1]: it suggests structuring the system feedback control loop into four components: Monitor, Analyse, Plan, and Execute. Several MAPE patterns for organising the adaptation logic in decentralised self-adaptive systems are described in [8]: although they are related and operate in a similar design context, they focus on internal organisation of system adaptivity, rather than on external application design. In particular, consider the *Regional Planning* pattern [8], which consists in distributing *Planning* components to different "software regions" (i.e., loosely coupled software subsystems); where they collect data from *Analyse* components (which are fed by *Monitoring* components) and command *Execute* components for enaction of planned adaptations. SCR somewhat subsumes Regional Planning, from the point of view of scope: it goes beyond the design of self-adaptation control loops by covering various assignments of responsibilities to the participants and being directly useable for application logic as well; e.g., leaders in SCR may gather regional data, resolve contention, or propagate events.

Similar to MAPE is the *Observer/Controller (O/C)* architectural pattern [49] for observation and control of organic systems. In the O/C pattern, an *observer* component collects information about a *System under Observation/Control (SuOC)*, and interacts with a *controller* component that is responsible of steering the emergent behaviour of the SuOC. In [50], variants of the O/C patterns are introduced, from fully centralised to fully decentralised designs. The SCR pattern is mostly similar to a dynamic combination of the distributed and multi-level variants of the O/C pattern, where the region is the abstraction that reifies a two-level structure, and the leader is the component at the upper level that evaluates and guides the self-organisation process at the lower level.

The *Multi-Scale Feedbacks* pattern [6] deals with large-scale coordination in hierarchical self-* systems. The pattern characterises a self-* system as a set of entities, exposing *observable* features, that are *associated with* or *composing* other entities. Then, it defines *micro-to-macro information abstraction* and *macro-to-micro feedback* as key functions between micro and macro features. Even though SCR can be applied to hierarchies and hierarchically, and shares some similarities with Multi-Scale Feedbacks – e.g., *inter-level feedbacks* (*downward/upward causation*) are comparable to SCR downstream and upstream information flows, assuming leaders are at a higher level than other agents –, the two patterns have different goals and take different perspectives: while SCR focuses on coordination of decentralised activity and interactions, Multi-Scale Feedbacks focuses on hierarchical design.

The SCR pattern also shares some similarities with *clustering*, the process devoted to the creation of *clusters*, i.e., groupings of "similar" objects—typically for data analysis and statistics, automatic classification, and community detection. Indeed, the process of formation of regions does include a clustering process,

Table 1
Examples of specialised terminology for the pattern components in different contexts. Some of the terms come from the known uses of the pattern summarised in Section 2.5.

Pattern term	Synonyms/specialised terms based on context				
	Networks	*Master/Worker Architecture*	*Cluster Management*	*Coordination*	*Others*
Leader	Hub, Root [51], Cluster Head [52–54]	Master, Control plane	Manager	Orchestrator [9], Coordinator [55]	Principal, Supervisor [56], Organiser [57], Centroid
Candidate leader		Secondary master	Backup manager		
Member	Node, Leaf [51], Sink [15]	Worker, Slave	Agent	Component, Coordinable	User, Follower, Subordinate [56], Member, Participant
Intermediary	Relay [10], Gateway [54], Link, Router	Work queue		Channels, Connectors	Forwarder, Middleman, Mediator
Region	Partition, Cluster [53], Community	Subtask	(Sub-)Cluster	Team, Coalition	Area [55], Division, Subsystem, Group

where leaders can be thought of as *centroids*. A region, in fact, could also be called a cluster. However, the SCR pattern also specifies responsibilities and dynamical processes within a multi-clustered system (see Section 3 for details), where each cluster is to be regulated through a feedback loop.

Other related abstractions can also be found, e.g., in the context of WSN research. For instance, the notion of *directed diffusion* [58] models data propagation and aggregation through localised interactions; the *abstract region* [59] abstraction captures various communication patterns within sensor regions; and *logical neighbourhoods* [60] support the definition of *virtual sensors/actuators* as well as localised interactions, through routing protocols that enable a notion of neighbourhood that goes beyond physical proximity. Logical neighbourhoods leverage neighbourhood templates matching node attributes, similarly to *attribute-based communication* paradigms [61]. Recent surveys [62,63] cover these and other related abstractions, aimed at managing dynamic groups (also called *ensembles*) of interacting entities, that may be adopted in implementations of the SCR pattern to deal with network-wide connectivity and communications.

2.5. Known uses

Though in this paper we describe a novel conceptualisation into the SCR pattern, it is interesting to discuss various forms and uses of it that can be found in the literature, and which justify its introduction as a *pattern*. Since terminology can vary (see Section 3.10), we have indicated the terms used throughout this paper as well as common synonyms and mappings in Table 1.

Decentralised service orchestration. In [9], SCR is used to design a decentralised service orchestration system; there, a workflow specification is split for scalability and performance into sub-workflows executed by multiple collaborating engines that are migrated to different network regions based on placement analysis.

WSN algorithms and middlewares. TCMote [14] is designed according to SCR. The system is organised in (possibly hierarchical) *sensor regions* governed by *leaders* with higher capabilities than the other region nodes (called *motes*). TCMote uses tuple channels for one-to-many and many-to-one communication between region sensors and the region leader in a single-hop. In another WSN middleware, *TS-Mid* [15], tuple space-based logical regions are used for power saving; there, regional leaders dispatch operations to normal nodes and transmit results to sink nodes. Moreover, several works consider clustering and routing in WSNs. In [53], a dynamic clustering scheme is proposed for self-configuration of WSN nodes, with the goal of increasing energy-efficiency by optimising the sleep times of nodes within each cluster based on traffic. The clusters (i.e., the regions) are formed by having nodes choose their *cluster head* (i.e., their leader) based on closeness with respect to a signal strength metric. In [52], the K-medoids algorithm is combined with affinity propagation to identify a set of cluster heads (i.e., leaders) in order to support energy-efficient routing in WSNs. In [54], a two-layered control scheme is introduced to support independent clustering and routing. Cluster heads collect and process data sensed from cluster members and send data to a different cluster. Interestingly, this work considers overlapping regions (see Section 3.9) where *gateway nodes* provide support for inter-cluster communication.

Robot swarm control. In the swarm steering study described in [13], the authors leverage dynamically selected, human-controlled leaders to influence and guide robot swarms towards dynamic goal regions. In [57], a distributed feedback mechanism à la SCR has been used to regulate wall construction: in this approach, a mobile organiser robot leverages a light source to provide a "spatio-temporal varying template" for the construction to be made; builder robots forage and deposit building materials, and may use flash signals to indicate "frustration" in the building process, causing the organiser to move. In [64], a SCR-like design with implicit leaders is used for formation control of UAV (Unmanned Aerial Vehicle) swarms. In [65], a methodology for complex multi-group coordination control is proposed, covering inter-group and intra-group formation (the latter adopting a leader–follower architecture), and leveraging a notion of *adaptive interactive force* to deal with inter-group interaction.

Traffic light control. The framework [66] for decentralised traffic light control is based on hierarchical multi-agent system organised as per SCR. *Region agents* model regions of the traffic network. They consist of *intersection agents* (SCR leaders) that coordinate with other intersection agents and control a set of *turning movement agents* (SCR downstream process), which learn to behave collectively and provide feedback at the corresponding intersection (SCR upstream process).

Resource management. In [67], a hierarchical system for the management of resources in large-scale multi-agent domains is described. In this approach, called *Distributed Dispatcher Manager (DDM)*, agents are organised into teams where communication is restricted to happen only between group subordinates to the corresponding team leaders. Leaders, which can also be grouped to form higher-order teams, collect information from subordinates and propagate resource assignments back. In another work, *Mission-oriented Adaptive Placement (MAP)* [68], SCR is adopted to implement a resource management framework for self-adaptive dispersal of computing services in multi-layer infrastructures. The approach leverages regional load balancing and inter-region coordination for global load balancing.

Decentralised reinforcement learning. In [56], a decentralised approach to reinforcement learning is proposed that leverages a *multi-level, supervision-based organisation* to coordinate the learning process: there, lower-level agents (called *subordinates*) are grouped into clusters, depending on how much they interact together, and report states and rewards to supervising agents (called *supervisors*), which in turn provide supervisory information to guide the learning agents in the exploration of their state–action spaces.

Morphogenesis and nature mimicry. In [51], SCR is used to implement an algorithm, inspired by the working of vascular systems of plants, for the dynamic distribution of resources aimed at the regulation of morphogenetic processes. This is called *Vascular Morphogenesis Controller (VMC)*. A VMC system consists of an acyclic directed graphs where root nodes (i.e., the leaders) acquire and distribute resources to leaves in a *forward flow*, and leaves, depending on their environmental conditions, provide a *backward flow* of guiding signals used to adjust the thickness of connections—influencing the amount of resources flowing in, which in turn affects creation and removal of nodes (cf., branching and shedding in plants).

Other known uses. Instances of the pattern can be found in other works that include distributed sensing [69], target counting [17], group management for target tracking [16], situated problem solving [55], design of self-adaptation control loops [8] (as discussed above), crowd tracking and steering [11,12] in opportunistic IoT, as well as peer-to-peer clouds [10].

3. Self-organising coordination regions

Self-organising Coordination Regions (SCR) is a particular way of designing systems that are made of multiple, heterogeneous (by constitution, position, or role) components that must organise their activity in a logical or physical space. Since it often recurs in the literature (see Section 2 for a detailed account), it is, by definition, a *pattern*, which we describe in this section in a general and systematic way (roughly following classical schemas for pattern presentation), while also exploring and documenting its variety of forms. The pattern builds on a space-based *divide-and-conquer* principle, and organises structure and interaction to enable self-integration while providing a trade-off between simplicity, flexibility, and efficiency.

3.1. Intent

The intent of the SCR pattern is to promote, in a self-organising fashion, the formation of dynamic groups of components, while taking into account the context (as induced by the environment or problem space), as well as to regulate individual- and group-level decision making.

3.2. Context

The pattern applies to *distributed* systems, possibly composed of *heterogeneous* devices, in which neither *full centralisation* nor *full decentralisation* in control and decision making is possible or desirable. Often, full centralisation introduces a single point of failure and reduces scalability, as such, it can be undesirable for *large-scale* systems, or even for systems that start small but need to dynamically scale up by need. Full decentralisation is inherently complex in achieving consensus and globally optimised functions; this is especially true in case devices are heterogeneous. For more discussion about the situations in which the pattern is most useful or applicable, refer to Section 3.6.

3.3. Name and synonyms

The pattern has been given the name *Self-organising Coordination Regions*. This reflects the decentralised nature of the pattern, as well as its support for coordination through scoped, endogenous, emergent structures and dynamics. Another suitable name could be *Decentralised Multi-Orchestration Loops*, as the pattern defines a decentralised coordination strategy for injecting multiple orchestration points into a system, creating corresponding system partitions regulated through feedback loops. In [70], the pattern was named *SGCG*, to denote the chain of aggregate programming blocks that provides a possible implementation schema of the pattern (see Section 4). Following the Leader–Followers distributed control pattern, another suitable name could be *Leaders–Followers*, where the aspect of regional organisation is kept implicit.

3.4. Structure and participants

Structurally, the pattern is organised as of Fig. 1. The system can be logically represented as a network of *nodes* situated in spatially extended and possibly dynamic structures called *regions*. These components can assume at any time one or more of the following roles[1]:

- *Candidate leader*: a node that is eligible, by virtue of its position, resources, or capabilities[2] for being elected as leader of a group of nodes or a region of space;
- *Leader*: a node, responsible for a region, which collects information from other nodes in its region and enacts decisions within the region;
- *Member* (of a region): a node (e.g., a user or worker node), subordinate to a leader, that sends/receives information to/from the leader of the region it is part of, possibly through intermediaries;
- *Intermediary*: a node that mediates interaction between leaders and members.

The regions may be logical or physical, may cover a part or the entirety of the space, and may be strictly separated or overlapping. The intermediaries mediate the interaction between leaders and members; sometimes, e.g., in peer-to-peer networks, these may work as relays. Also, note that the two-layer organisation of the pattern (into leaders and subordinates) is orthogonal to the (possibly flat or many-layer) structure of the underlying network of devices (cf. the case study in Section 5.4).

[1] Depending on the scenario and the particular instantiation of the pattern, the types of entities involved may take specialised names, such as those reported in Table 1.

[2] Even though the pattern itself makes no assumption on the network structure, on an edge deployment usually candidate leaders correspond to edge servers.

3.5. Dynamics and collaborations

The pattern is organised in four phases (graphically represented in Fig. 2):

1. *Election of leaders*. Leaders are elected from the set of candidates. They will be in charge of coordinating the region members, e.g., by planning or taking decisions.
2. *Formation of regions*. Each node is assigned to a single leader, thus creating regions.
3. *Information flow from members to leaders*. Member nodes stream data or updates regarding their local activities or perceived events to leaders; some processing (typically data aggregation) can occur *en-route*—examples include sensor data, local events, service requests, or feedback information for the assigned tasks.
4. *Information flow from leaders to members*. Leaders stream computation results to all members of their managed region—it may be a decision to be enacted, a collective view to be propagated, instructions to be assigned, and so on, depending on the use case.

Note that these phases, although presented sequentially, are actually dynamic processes that happen concurrently and are related by input/output dependencies (see Fig. 3). In particular, the leader election phase can be thought as an active process that can react to perturbations by automatically revising the selection of leaders (i.e., in a self-healing or self-adaptive way), consequently causing a reshaping of regions (therefore, we can say that the pattern promotes self-organisation). Then, as regions change, the processes of upstream communication (collection) and downstream communication (propagation) need to adapt to the new situation as well. Additional flexibility comes from the possibility of equipping the system with feedback loops: information propagated by leaders may induce an effect on workers that, in turn, is perceived by leaders through collected data. This feedback may induce the system to self-configure itself, e.g., in order to fine tune the coarseness of the regional structure.

So, in its most general form, with a dynamic set of candidates and participants, and no assumption on devices' capabilities or network structure, the pattern requires and promotes forms of self-organising coordination, hence the "self-organising" in the pattern name. Nevertheless, it is possible to add assumptions on the structure of the system and specialise/optimise the pattern to tackle the very situation at hand. These pattern specialisations do not alter the self-organising nature of the pattern, though. Having self-organising and non-self organising variants of the same patterns and algorithms is in fact pretty common; as an example, consider the well known Bellman–Ford algorithm [71,72]: besides the classic versions working on a graph, there exist self-organising versions meant to operate on a distributed and dynamic setting [42,45]. Notably, variants supporting self-organisation can usually work in a superset of the scenarios supported by their non-self-organising counterparts.

3.6. Applicability: when to apply

The SCR pattern is encouraged in face of large-scale and situatedness, openness, requirements of data locality (e.g., because of privacy or latency or energy use), and balanced approach between centralisation and distribution. In particular, in many cases large-scale situated systems need to self-organise in such a way that its components can be monitored and coordinated according to a view larger than local (such as in complex situation recognition), but cannot take the risk of centralising in a single point of failure, nor can weigh down the whole coordination effort on a single device. This is particularly challenging in open systems, where leader candidates and nodes can join and leave the system dynamically, the underlying network structure is unknown and changing over time, and failures are likely. In fact, under these circumstances, the system may need to dynamically switch among configurations with a different count of centralisation points, which rules out full centralisation while at the same time making full decentralisation over-complex for low load situations.

Another key situation in which SCR finds natural application relates to heterogeneity. In some systems (e.g. mixed edge-cloud deployments) differences among devices reflect to different roles that the device can assume in the coordination process. For instance, end devices may feature constraints (battery use, data rate limits, computational capabilities) preventing their participation as coordinators of the system or one of its parts.

3.7. Applicability: when not to apply

Systems for which fully centralised or decentralised coordination is applicable are not good candidates for SCR application. In these systems, forcing the application of the pattern would result in degenerate cases. For instance, in a system where decision making can be decentralised entirely, forcing the adoption of SCR would degenerate to electing every node as the leader of a region containing solely itself hence introducing unjustified complexity and coordination cost. Similarly, on the opposite side of the spectrum, if centralised coordination is either possible or necessary, and there are no issues of openness and resilience (hence, the selected single leader is guaranteed to be available), SCR would represent an element of unnecessary complexity. In fact, even though in principle SCR can be used to elect a single leader and establish upstream and downstream communication, more effective options are usually available when the assumption of dependable central leader holds.

The problem at hand must be partitionable into sub-problems (regions) and solved independently. Hierarchical variations of the pattern can provide support for more complex cases, yet if the problem does not allow for partitioning, the pattern cannot get applied but for the degenerate variant in which it is used to elect a single leader. Other coordination patterns are recommended in these cases.

An important element to factor into the choice of whether or not to apply SCR is dynamicity. With respect to approaches using full decentralisation, SCR introduces a coordination overlay. If the system changes so quickly that, for instance, leader election cannot be performed, or that up- and down-stream communication channels cannot get established, then SCR could not be leveraged, and different approaches are to be applied. In particular, for the pattern to work, it is required that it is possible to elect a single leader per region. In case this is not possible, e.g. due to missing support for such an operation from the implementation platform, SCR cannot be applied. Moreover, until the region leader is elected, the outcome of the computation is undefined: strict deadlines or guarantees over the transient dynamic of the pattern may make it unsuitable for the scenario at hand.

3.8. Consequences

Hybrid decision making. Decisions are taken considering a tunable subset of the whole system, de-facto creating a hybrid between centralised and decentralised decision making. The amount of decentralisation depends on the granularity of the partitioning into regions: the more regions are created, the more control is decentralised.

Reduced dependence from deployment and network structure. SCR creates a sort of dynamic, adaptive network overlay structure on top of the existing communication infrastructure. By merely organising application logic on that overlay, the specific shape of the underlying network can be abstracted away, allowing for easier porting to diverse setups (e.g. cloud, edge, purely P2P), as show in Section 5.

Eventual consistency. Temporal mobility, loss of messages, and device failures, only temporarily affect the values collected in leaders, and hence, deviation from the actual global view. This effect is more pronounced as more intermediaries mediate interaction between leaders and subordinates.

Sub-network isolation. In most versions of the pattern found in the literature, members belonging to different regions do not participate in the same sub-system (i.e., they do not exchange information); however, indirect communication may be implemented through inter-regional communication between leaders. We note that sub-network isolation is a common but not necessary consequence of the pattern application: in few cases, extended versions of the pattern are presented that allow for inter-region communication (region overlapping) [70].

3.9. Variants and extensions

Pre-established leader candidates. Heterogeneity could in some cases force the set of candidate leaders to be predetermined, since part of the participants cannot assume the role of leader. This assumption usually reduces the search space for leader election, leading to quicker convergence. This variant can actually be seen a special case of the pattern application.

Leader election with pre-established regions. In certain contexts, the regional structure might be determined *before* leader election is performed, effectively inverting the order of phases 1 and 2 (see Section 3.5). This could be desired when the problem space has natural boundaries. Such *inversion of control* should simplify the task of leader election, by limiting the scope of that process to each specific sub-region.

Interconnected leaders. In some scenarios, channels among different leaders can be set up to enable more global, system-wide coordination and tackle needs going beyond those of individual regions. The connected leaders make up a group used to define higher-level regions, leading to a kind of hierarchical organisation. This variant is connected to Observer/Controller patterns as well as to multi-agent organisational paradigms, as covered in Section 2.4. Notice that having multiple leaders interact is expected to introduce integration conflicts [39], to be tackled, e.g., through collaboration, negotiation, and consensus.

Hierarchical organisation. The pattern is designed around a two-layer organisation (with leaders and followers). However, it can be applied recursively: a region can be split into sub-regions governed by sub-leaders, and so on. This approach can be taken, e.g., to organise decision making at a finer granularity, or to deal with problems existing at different scales. Organising SCR hierarchically enables multiple levels of information aggregation and multilevel control. This may turn useful in case a problem cannot be partitioned and resolved by a single leader, but the solution only relies on aggregate data from subsections of the whole system.

Overlapping regions. When regions overlap, then a node can be part of more than one region, which means that it could be required to follow more than one leader. Overlapping regions can be created by *not* imposing the constraint that a node must follow exactly one leader. Therefore, this variant introduces the issue of *arbitration*, i.e., of how to resolve conflicting commands issued by different leaders. When regions are created through distance-based gradients (see Section 2.3), overlapping regions can be achieved by spawning one gradient process (representing a region) per leader. Additionally, when regions are sustained by concurrent process instances, the phases of upstreaming and downstreaming are naturally scoped per process, namely per region, promoting isolation of regional activities. However, nodes that are members of multiple regions are in a privileged position (together with those at the regional boundaries) to support inter-regional communication and coordination.

Non-covering region set. In this variant, regions do not necessarily form a partition of the space. This could also be thought of as a partition consisting of $n - 1$ leader-regulated regions and 1 *free zone* (i.e., a leader-free region). The free zone may also be non-contiguous, meaning that there could actually be multiple separate free zones. Such an approach could be used to promote isolation between the regions.

3.10. A note on terminology

Depending on the context or domain in which the pattern is used, or on mere linguistic choices during design and analysis, the participants and other structures (cf. Sections 3.4 and 3.5) of the pattern may be referred to with specific names—such as those suggested in Table 1.

We use terms "leader" and "member" to stress that the system has two different *kinds* of entities, with *asymmetry* in their responsibilities, and that suitable entities ("candidate leaders") may be promoted as leaders through a proper dynamic process (that might depend on contingencies of the situation). These terms, indeed, denote *roles* that various types of entities may play when viewing a system by the perspective of the pattern.

We also introduce an entity called "intermediary" to capture aspects related to communication between leaders and members. Indeed, members may not be directly connected to leaders. In a network, such as a WSN or a peer-to-peer network, an intermediary could be responsible merely of routing messages, and hence may better be called a "relay" or "router".

We use term "region" to denote *a priori* or *a posteriori* structures where leaders and members work. Depending on the application, this might denote a "team" or another kind multi-agent organisational structure [34], be a "partition" (if a member belongs to exactly one region), or generally be a "subsystem".

For instance, in [55], the SCR pattern is used to organise an ecosystem for situated problem solving, which could be instantiated, e.g., in a smart city maintenance scenario. The idea is that problems that arise are *situated* (i.e., they have a location in space–time) and have to be found and solved in place. Leaders are called "coordinators", to stress their role in coordinating the problem solving activity (coherently, regions are called "coordination areas"), whereas members are called "agents", to stress they are largely autonomous and that potentially include both (augmented) humans and robots. Agents come in two flavours, "detectors" (responsible for detection of issues or problems to be solved) and "solvers" (responsible for fixing detected issues). In this scenario, upstream data includes "detected problems" and "capabilities" of the solvers, whereas downstream data includes "assignments" of problems to solvers.

4. Implementation

In this section, we discuss the implementing platform requirements to support the pattern, we describe some possible variants in the implementation strategy of the four phases described in previous section (Section 4.1), and then provide a prototypical implementation strategy in the framework of Aggregate Computing (Section 4.2).

4.1. Implementation issues

4.1.1. Election of leaders and formation of regions

The goal of the election phase is usually a configuration of leaders that must be valid or optimised with respect to a particular property—e.g., uniformity in spatial coverage (as of a smart city environment) or balancing of load (tasks, workers). Consensus on leadership may involve centralised algorithms, or resort to (more challenging) algorithms for distributed and asynchronous systems [73,74]. Generally, not all devices are eligible for leader position. Indeed, there could be constraints or preferences concerning which nodes can be selected as *candidates* for leadership. For instance, a system composed of both mobile and static (non-mobile) nodes may restrict the set of candidate leaders to the latter category of devices, e.g., because they are connected to a power outlet. In general, in heterogeneous systems where devices feature very diverse resource availability and constraints, coordinators are preferably dependable nodes with significant computational and network resources, and little or no power saving concern—such as edge gateways or fog nodes. The choice is driven by the will to minimise the cost for the leader election process, which is likely to be less expensive if the elected leaders remain the same over time, hence promoting an upstream selection of candidates less likely to leave the system. Trust could also be used to rate and therefore include/exclude nodes from the candidate set based on observed activity [75], reifying a notion of trust community [76], whose deeper investigation is left as an interesting future work. The election process can be part of the system bootstrap process or be dynamically reconsidered, continuously or after a delay. The latter case is more common in the scenarios where SCR is usually found, as often mobility, failures, or functional changes in the operational context (e.g. load peaks, new sensor readings, unexpected situations) may require the invalidation of the current leader configuration.

The association of a component to a certain region is generally a consequence of leader election, since components tend to follow their closest (according to an arbitrary metric) leader. The problem is similar to *membership* in component-based *ensembles* [77], which is sometimes tackled via *attribute-based coordination*. Another way to define a region (also used in the implementation schema of Section 4.2) is based on the construction of a gradient (see Section 2.3) having the leaders as sources. Since each node computes the distance from the closest leader, attaching its identity to the propagating information provides for the creation of regions contextually with the propagation itself. The same technique, also known as gradcast [78], can be used for spreading in the same way arbitrary information—see Section 4.1.2. This propagation technique can be particularly valuable for implementing SCR, as it natively introduces a directionality structure, which can be leveraged not just for propagation, but for collection (upstreaming) as well—see Section 4.1.3. When relying on gradients, the case of multiple, overlapping regions, where components may participate to multiple collectives, requires special handling, as natively a gradient propagated from multiple sources has no overlaps, but simply refers to the closest source. To cope with this situation, a different gradient needs to get propagated from each source. When the count of possibly overlapping regions is dynamic and only known at runtime, abstractions for concurrent collective computations [70] may prove useful.

4.1.2. Information spreading (downstream)

When SCR is deployed, the most common ways of disseminating information are gossip [79] and gradient-based information cast, also known with the short name of gradcast [78].

Gossip protocols [79] are suitable for letting information flow from leaders to members, assuming that information is monotonic, i.e, that it can only change in a single direction, as is the case, for instance, for timers, whose values can only grow. This limitation however severely restricts the application range of pure gossip, since many interesting use cases need to manipulate data that changes in any direction, or for which it is not even meaningful to define ordering. Whenever information is not monotonic, gossip algorithms should get periodically refreshed or reset (thus introducing perturbations), or overlapping replicates of the algorithm should execute in parallel [80].

Gradient-based information spreading algorithms are based on the idea of carrying information along with a monotonically-increasing (logical or physical) distance from the information source. They are suitable both for generating regions once leaders are elected (by propagating a gradient from each leader and selecting the closest) and for disseminating information from leaders to members, implementing a downstream information flow. Several implementations of the algorithm exist, ranging from distributed adaptive Bellman–Ford [44] to advanced versions and compound algorithms taking into account aspects like time, speed, and acceleration of devices [42,78,81].

4.1.3. Information accumulation (upstream)

When no assumptions on the network structure can be made, information accumulation is generally a tougher task than information spreading. As for spreading, accumulation can be realised by gossipping information such that the leader is reached with messages from all nodes in the region: however, this effectively works only in the case of small regions, as it usually involves a considerable amount of resource utilisation. Moreover, the same limitations of monotonicity of information discussed in Section 4.1.2 for the spreading hold for accumulation as well.

A more scalable technique is based on building a spanning tree over the network (locally selecting as parent the closest neighbour to the source), then accumulating along such tree towards the leader. Spanning trees, however, are highly fragile to changes in the network [82]: disruption and creation of links may lead to different configurations, making naive versions of this algorithm unsuitable for scenarios with frequent network reshaping [37]. In these cases, multi-path techniques aggregate information towards the source using multiple spanning trees rather than a single one. They are usually more robust to changes in the network structure, but take more time to converge in case of stable networks [37,83].

4.1.4. Platform requirements

Actual implementations of design patterns can vary significantly depending on the language or platform (more generally, the *substrate*) meant to be used to implement them. Sometimes, the evolution of substrates leads to easier implementation or even direct support for a pattern. To better explain the concept, we take as example the original work from Gamma et al. [18], whose pattern were exemplified in C++. We note how almost a decade later, some of the patterns could be implemented straightforwardly using aspect oriented programming, e.g. with Java using AspectJ [84]. More recent languages embed common patterns into language constructs: for instance, Scala [85] offers language support for creating singleton objects; and Kotlin provides a `by` keyword to implement the delegation pattern. In short, different substrates mandate different implementation strategies for the same pattern, and SCR is no exception.

In particular, substrates offering easy means to:

- partition devices into regions;
- elect a leader within a partition;
- diffuse information; and
- aggregate information to a sink

are good candidates for quick and easy implementations of the pattern. On the other hand, substrates which entirely forbid any of the aforementioned operations cannot enjoy SCR, but possibly can for some of its less interesting variants (e.g., with a statically defined set of leaders, with no election at runtime, and hence with reduced resilience to leader loss). One key requirement is the ability to define regions in an either physical or logical space, which implies the ability to measure distances in such space. The metric is relevant for the effectiveness of the pattern, as regions built with metrics not mapping any relevant characteristic of the problem at hand may end up producing aggregates which cannot achieve, or cannot achieve *effectively*, the coordination goals. The illustrative implementation proposed in this paper in Section 4.2 conveniently picks one framework providing simple means to implement the constituent parts of the pattern. Other frameworks in the research realm of organic computing, such as DEECo [86], and network abstraction languages (see Section 2.4) such as Logical Neighbourhoods [60] or SCEL [77] are in principle well suited to support the pattern as well.

4.2. Illustrative implementation

We propose an implementation schema for the pattern in the paradigm of aggregate computing [11,87]—used in next section as a basis for evaluating a smart city case study. The reason for this choice is rooted in the rather straightforward mapping between the sub-patterns of SCR and the building blocks available in existing aggregate computing languages, which allow for a concise implementation.

4.2.1. Background: computational fields and aggregate computing

Aggregate computing is rooted on the idea of programming distributed systems from a global perspective, declaratively [11], by functional manipulation of distributed data structures called *(computational) fields* (time-evolving maps from devices to values). The *field calculus* [87,88] is the formal, universal, minimal language for functionally composing and manipulating fields, based on which domain-specific languages (DSL) like ScaFi [69] and Protelis [89] have been introduced to specify, simulate and run self-organising behaviours and collective coordination logic. Similarly to organic computing [22], aggregate computing also aims to address robust local-to-global behaviour by steering emergence and balancing top-down control with decentralised, bottom-up processes.

In the field calculus, a program describes a collective behaviour by neglecting the single-device viewpoint. However, the operational semantics [88] defines how the single device can "continuously" process the program and sustain the overall system behaviour, by cyclic steps encompassing: *(i)* assessment of a local context (previous state, environment perception, collection of input messages received so far); *(ii)* interpretation of the aggregate program against such a context (producing a new state, messages to be sent, and actions to be executed); *(iii)* execution of actions and spread of messages to neighbours.

To show high-level examples of aggregate programming we consider ScaFi, a Scala-internal [85] DSL exposing the primitive constructs of the field calculus (as well as a library of higher-level functions). A detailed explanation of the field constructs goes beyond the scope of this paper; the interested reader is encouraged to refer to [88] (for the field calculus) and [90] (for ScaFi). As an example program to ground the discussion and grasp the key aspects of the programming model, consider the following definition of a `distanceTo` block (i.e., a self-healing gradient):

```
def distanceTo(leader: Boolean,
metric: ()=>Double) =
  rep(Double.PositiveInfinity)
     (distance => mux(leader)(0.0)
     (minHood(nbr(distance) + metric())))
```

In the ScaFi code to come, purple symbols are non-primitive aggregate building blocks, grey symbols are configuration parameters, and bold symbols denote methods for local activity to be tailored to the application. Function `distanceTo` specifies a global behaviour: it takes a field of Booleans indicating `true` on leaders and a `metric` associating a distance to any pair of neighbouring nodes, and expresses how to compute and returns a field mapping each node with the minimum hop-by-hop distance from the nearest leader—most specifically, by computing triangle inequality as detailed e.g. in [37,42]. Indeed: `rep(init)(f)` evolves continuously field `init` through function `f` (by a local operational perspective this is achieved by repeated execution of function `f` against the local value for `rep`'s field, or `init` when no such value is present—like on the first local execution); `mux(c)(a)(b)` uses Boolean field `c` to project either `a` (where and when `c` is `true`) or `b` (otherwise); and `minHood(e)` folds over each neighbourhood and selects the minimum value for expression `e`, computed by replacing `nbr(m)` with the neighbour's value for expression `m`. Just like any called aggregate function, `distanceTo` is to be interpreted by each node of the network in repetitive rounds, resulting in a continuous process of sensing the environment (to update the two inputs), receiving messages from neighbours (containing their latest value of `distance`), and broadcasting values to neighbours (the local value of `distance`)—according to the operational semantics reported in [37]. Most importantly, `distanceTo` can be functionally composed with other similar blocks to achieve more complex behaviours (like the SCR pattern), still resulting in the same sense-receive-broadcast(-act) protocol: each message will transparently carry all the required information to support the evaluation of each composed block.

4.2.2. Pattern implementation schema

We are now ready to show an implementation schema for the proposed pattern in ScaFi. Details on the syntax, as well as on the implementation of sub-patterns leveraged in the following code, can be found in [90]:

```
class SCR extends AggregateProgram with
BlockG with BlockC with BlockS {
  def main = {
    // selects a field of leaders, with
    // at least grain distance
    val leader = branch(isCandidate)
    { S(grain) } { false }
    // creates a gradient from leaders
    // based on a given metric
    val potential = distanceTo(leader,
    metric)
    // gathers localInput values towards
    // leaders by aggregation
    val convergeCast = C(potential,
    localInput, aggregationFun)
    // on leaders, takes a local decision
    // based on received data
    val decision = decisionMaking(leader,
    convergeCast)
    // broadcast decisions and take action
    val divergeCast = G(leader, potential,
    decision)
    localAction(divergeCast)
```

```
  }
}
```

This code is a Scala-based script representing both the overall collective adaptive behaviour, as well as the code to be locally executed by each device to sustain its part of computation of the SCR distributed algorithm. Function names `S` (from sparse-choice), `C` (from converge-cast), and `G` (from gradient-cast) are inherited from the original work introducing self-stabilising building blocks for aggregate programming [37,91]. These names are thus used in the proposed implementation. Function application `branch(c){e1}{e2}` performs domain partitioning according to condition `c` (i.e., devices for which `c` is `true` will run aggregate behaviour `e1`, otherwise `e2`): this is used to constrain the leader election process to the set of candidate leaders. Call `S(g)` activates a decentralised leader election process yielding a Boolean field (assigned to local variable `leader`) that is `true` in correspondence of active leaders [74]; in particular, this form ensures that leaders are at a mean distance `g` between each other. Notice that since the aggregate program is continuously run, a change in the leader set configuration (as induced by mobility or failure) would be handled reactively, possibly electing new leaders or transferring leadership. Call `distanceTo(s,m)` builds a self-healing gradient field (using metric `m`) from devices where `s` is `true`; this is used to make device compute an estimate of their minimum distance to the leaders, both for membership and for constructing a dynamic communication structure providing a direction for upstream and downstream information flows. Call `C(p,l,f)` aggregates local values `l` via function `f` along potential `p`: this implements an upstream information flow [83], where `l` and `f` are application-specific. Call `decisionMaking(r,d)` is also application-specific and depends on the role `r` of the device and the data `d` locally available from the upstream. This decision can change across time based on history and the change of inputs. Call `G(s,p,d)` propagates data `d` locally available from sources `s` along the direction provided by potential `p`: this implements a downstream information flow used by leaders to enact their decision. Finally, call `localAction(d)` is a local, application-specific behaviour based from the information downstream.

We stress that the provided schema is not just pseudocode but actual ScaFi code that can be executed on a (simulated or real) network of devices. We also observe that the schema does provide a structure for organising self-integration processes, whose specifics can be defined via a proper selection and definition of parameters, application-specific functions, and code extensions.

5. Case studies

In this section, we show two possible uses of the SCR pattern in different contexts a dynamic network of situated and mobile devices performing collective sensing, and a hierarchical networked system performing collaborative and dynamic task distribution. The former case shows how SCR can be used when there is need to integrate systems dynamically, and maintain operativity in contexts with high dynamicity. The latter case applies the pattern to a more classic network structure, showing its ability to perform on-the-fly online self-organising integration of subsystems depending on their dynamic state.

The goal of the evaluation is to showcase the suitability of SCR when self-integration is required to deal with changes in the system, due to mobility of users (in the first case) or changes in operating conditions (in the second case). The way we demonstrate the applicability of the pattern is by exercising it in simulation, and showing how the system is able to cope in a wide range of conditions. More precisely, we show how the pattern (if appropriately implemented) is self-stabilising [37], namely, to obtain correct behaviour no matter what initial state is given, and hence also to be able to attain correctness in face of a disruption (in our cases, user mobility and a peak load).

In both case studies, we evaluate specific instances of the SCR pattern. Of course, this implies that the extrapolated numbers are strictly related to those specific implementations, without claim of generality. However, evaluating these implementations helps also shedding light on the pattern at large: in particular, on the approach to adopt when dealing with variants using feedback loops (and the importance of appropriately control such loops, as learned from control theory); on the potential effectiveness in diverse contexts (situated collective systems and classic networked systems); as well as expectations about the overall dynamic of the SCR-controlled system. More in general, in this section we demonstrate that appropriate implementations of the pattern exist that keep the promises of self-organisation and self-integration made in the first part of this paper. Our experiments and selected metrics are intended to show that SCR (appropriately implemented) is:

1. resilient to changes of different nature (moving users in one case, a sudden peak load in the other);
2. robust to changes in the specific parameters that regulate its behaviour;
3. predictable in face of changes in control parameters; and
4. correctly reacting to transient conditions, i.e., continually operating in constantly changing conditions, without showing resonant or unexpected behaviour.

5.1. Background: Edge computing

Fog and Edge Computing [92–94] are emerging ICT paradigms that aim to bring cloud-like functionality closer to the edge of the network, i.e., to where end users and data sources reside (or, generally, to where computational intelligence is mostly required—cf., IoT and CPS). This is highly desirable especially in the following circumstances:

(i) *when the powerful, remote cloud is not accessible*, e.g., because there is no global Internet connectivity;
(ii) *when the remote cloud is accessible but it cannot satisfy quality attributes and requirements*, for instance because of issues in data privacy or the inability to set real-time guarantees due to high latencies in communication;
(iii) *when the remote cloud is both accessible and useful but expensive*, e.g., in terms of subscription fees or network bandwidth.

That is, Edge Computing is in some cases a necessity, but in general it complements remote cloud computing with a whole new set of possibilities ranging from infrastructure-level optimisations (from exploiting idle edge devices to filtering data before upstreaming it to the cloud) to flexibility in service-level agreements and robustness through decentralisation.

5.2. Implementation details, organisation, and reproducibility

This section summarises some details shared between the two cases. In both cases, SCR has been implemented in Protelis [89], which has been favoured over ScaFi (used in the examples presented in this manuscript for its more compact and familiar syntax) as it comes with a larger and well tested library [78], and, generally, with a more solid toolchain, allowing us to reuse well-tested building blocks in the experiments (which, of course, have to deal with a larger number of details). The implementations under test have been simulated using Alchemist [95], an extensible event-driven discrete event simulator. In Alchemist time is

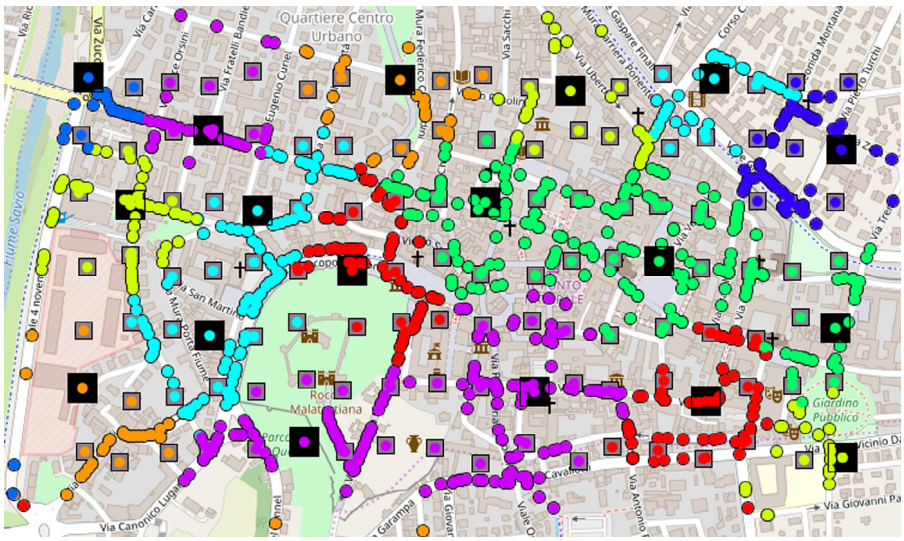


Fig. 4. A snapshot of the first simulation in execution. Every device is represented as a circle. Edge servers are also surrounded by a square, large and black for currently elected leaders, smaller and greyed for unelected leaders (working as intermediaries). Colour of the circles identifies the region the device is assigned to. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

not marked by discrete ticks, but is instead continuous, evolved by the succession of simulated events. It can be seen as a peculiar agent-based simulator featuring a set of optimisations derived from stochastic Monte-Carlo simulators; and has been used in a wide range of applications, ranging from morphogenesis [96] to social sciences [97] to IoT applications such as crowd tracking and steering [98].

For the first scenario, we executed 100 repetitions of the experiment for each configuration in the cartesian product of the parameters' values, varying displacement of edge devices, initial position of users and their waypoints, and execution times of devices. For the second scenario, we first executed 10 repetitions of the experiment for each configuration in the cartesian product of the parameters' values, and we used the data to appropriately select the self-integration algorithm parameters. Once these parameters got fixed, the search space was reduced, and we executed 90 further repetitions with the selected values. Data generated by simulation has been processed using Xarray [99] and plotted using matplotlib [100]. For the sake of reproducibility, the experiments have been entirely open sourced, and instructions have been provided for re-executing them. The code includes a reference implementation of the SCR pattern, all the material has been released on publicly accessible repositories.[3,4] For the sake of conciseness and clarity, this paper analyses and discusses only a selection of the whole data generated by the experiments and available on the provided sources.

5.3. Resilient multimedia streaming on the edge

As a first scenario, consider multimedia processing contexts that involve transmission and computation of user-generated video streams. Real-world application examples may include, for instance, multi-view media generation and fruition (e.g., for multiplayer gaming) [101] and metropolitan collaborative surveillance [102]. In the past years, pervasive usage of multi-view and 360-degree-view video streams was largely discouraged by scarce tolerance to delays and huge bandwidth requirements and usage [101]. However, telecommunication technology advances (cf. 5G and 6G) are mitigating the latency and bandwidth issues. Additionally, relevance of low-latency video processing will likely increase in the future thanks to applications like mobile augmented reality [103]. The execution of such multimedia applications is to be supported across the smart city environment, where users wearing mobile devices (such as smartphones, or even augmented-reality gadgets) can move and interact with cyber–physical components. The smart city infrastructure consists of a network of static (non-mobile) edge servers, with which mobile devices can communicate. The goal of the overall system is to self-organise to dynamically select a subset of edge nodes (enough to sustain the computation) to work as local leaders, integrate user devices and other infrastructural components into leader-regulated teams, collect and redirect the video streams from users to the corresponding leader edge device, process the data gathered at the leader, and finally spread downstream the result of the computation back to the users.

5.3.1. Experimental configuration

Concretely, our scenario is composed of multiple edge servers (specifically, 126) deployed in the centre of the Italian city of Cesena, participating the system as leader candidates. They are displaced as an irregular grid, to emulate the physical limitations of a real-world deployment (see Fig. 4), and their exact position on the map varies across simulation runs. We dynamically elect a subset of these candidates to work as leaders, and let the others participate the system as relays. More precisely, edge servers elect a leader for every region with a radius of 200 m, competing using the S building block (namely, leveraging unique identifiers to break symmetry, and preferring established leaders to novel ones if in range, promoting stability).

The system's goal is to collect data streams generated by users, aggregate them, and diffuse the number of streams being processed to the whole region. Users are modelled as devices moving along roads open to pedestrian traffic at a constant speed of 1.4 $^{m}/_{s}$. For detecting streets open to pedestrian traffic, we rely on data obtained from OpenStreetMap [104]. Bidirectional communication is considered established between users and edge servers, and among edge servers, if physical distance is within Wi-Fi range (100 m). Users do not directly communicate with each other. In this work, we do not consider signal attenuation due to objects along the line-of-sight of Wi-Fi antennas, nor we

[3] https://bitbucket.org/danysk/experiment-2019-coordination-dynamic-orchestration

[4] https://github.com/DanySK/Experiment-2019-FGCS-Self-Integration

Table 2
Free variables for the first scenario in exam.

Name	Description	Values
u	Active user devices count	[50, 100, 200, 500, 1000]
α	Backoff algorithm parameter	[0, 10^{-3}, 10^{-2}, 10^{-1}, 1]
ρ	Probability for a leader to shut down after 10 min	[0, 0.25, 0.5, 0.75, 1]
fb	Determines whether the feedback loop is enabled	[true, false]

consider bandwidth capacity and data rate reduction due to concurrent access to the shared communication medium. We deem a deep analysis of network details to be out of the scope of this work, which aims instead at showcasing SCR uses, and drawing conclusion that can shed light on the ability of the pattern to cope with continuous change. In fact, an evaluation leveraging a more realistic network model, including details such as signal propagation and attenuation, protocol overheads, data rate reduction due to shared resources, and so on; would be incredibly useful to have insights on a system prior to its actual deployment; but would not add many valuable insights on the *overall behaviour* of SCR in general—which is instead the primary goal of this work. To this end, we adopt a simplified network model, assuming connectivity with the edge servers within 100 m from the end device position. Our experiment timeline is the following: the simulation begins with devices bootstrapping asynchronously; after 10 simulated minutes, we simulate a disruptive event: elected leaders suddenly fail with probability ρ—e.g. as would happen due to a city-wise power shortage; after further 10 min of system evolution, the simulation ends.

We compare a classic implementation of SCR pattern (as described in Section 4) with a variant featuring a feedback loop. The latter, whose implementation schema is provided in the remainder of this section, tries to coordinate the leaders in such a way that they resize their regions in the attempt to cover approximately the same number of users, so as to reduce disparities in workload that could cause slowdowns on overloaded edge servers. This self-organising adaptation of region size is achieved by feeding back to the leader the count of users being currently served, and using it in turn to decide how much to compete with other leaders, with the idea that the more users are being served, the weaker is the leader "claim" on its region, and hence the smaller is the resulting region area. Feeding the served user count back to the algorithm input directly, (as any unregulated control loop) may lead to oscillating and possibly resonant behaviours, which we want to prevent. Hence we filter it using an exponential backoff (a low pass filter), namely, the feedback value is $\alpha u_t + (1-\alpha)u_{t-1}$, where u_t is the count of served users at time t, starting from 0 clients being served at bootstrap . The α parameter is the sole non self-tuning part of the implemented instance of SCR, and requires manual tuning. Impact of feedback loops on the actual implementation may vary widely, depending on the specific feedback being implemented and on how easily the platform hosting the SCR implementation can accommodate changes. In our case, changes w.r.t. the classic implementation proposed in Section 4 were minor. A code snipped exemplifying the feedback loop introduction is as follows:

```
class SCRFeedback extends AggregateProgram
                  with BlockG with BlockC
                  with BlockS {
  def main = {
    val leader = branch(isCandidate)
    { S(grain) } { false }
    rep(0) { clients =>
      // low pass filtering via
      // exponential backoff
      val filtered =
        exponentialBackOff(clients, alpha)
        // creates a gradient from leaders,
        // setting the client count
        // as bottom value, hence
        // restricting the areas for leaders
        // serving a large count of clients
        val potential = distanceTo(leader,
        sourceValue = clients, metric)
        val convergeCast = C(potential,
        localInput, aggregationFun)
        // upstream aggregation counting
        // the clients being served
        val currentClients = C(potential,
        1, _+_)
        val decision =
        decisionMaking(leader, convergeCast)
        val divergeCast = G(leader,
        potential, decision)
        localAction(divergeCast)
        currentClients
    }
  }
}
```

The most relevant change w.r.t. the code in Section 4.2.2 is the `rep` call wrapping the pattern. This primitive is meant to retain state across computation rounds, and it is hence necessary to keep track of the number of clients being actually served. In control theory terms, it is required to implement the integral part of the proportional–integral controller represented by the exponential backoff.

We first search for good values for α in our scenario, by looking at how different values affect the size of regions and their stability. We stress that α is the only parameter that requires manual tuning, as it is related to the specific kind of low-pass filter adopted for this case study. We then measure performance and resilience for both the base and the optimal-α versions of SCR, varying the number of users and ρ, and observe how many users are served overall and by each edge server. A summary of the free variables for the case study is given in Table 2; measures are instead summarised and explained in Table 3.

5.3.2. Results

We initially measure the impact of the feedback system and how it behaves with different values for α. Results are depicted in Fig. 5, detailed analysis is provided in the figure caption. They show that, among the analysed values, $\alpha = 10^{-2}$ works best.

We then evaluate correctness and performance of the algorithm both without and with feedback enabled ($\alpha = 10^{-2}$). Fig. 6 shows that the system is able to serve all the users, actually overprovisioning some of them, since the handoff between regions is not dealt with gracefully with an explicit session termination: hence, nodes crossing the boundary between neighbouring regions may happen to be served twice. This is not an issue for this specific scenario, as it actually provides some resilience in case of movements along the border of the region, in case the client is only temporarily transiting to a new region. It might however be a concern for applications with hard requirements on the handoff of nodes changing region, and highlights a more general issue

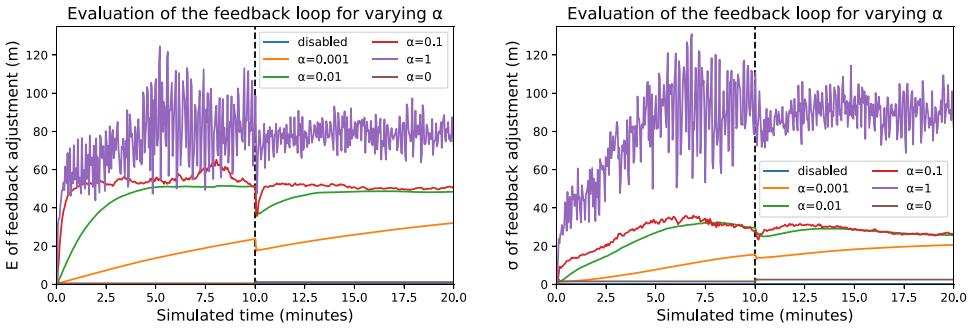


Fig. 5. Evaluation of the backoff parameter α, which tunes the trade-off between reactivity and stability. Values are averaged along all values of u and ρ. Setting $\alpha = 0$ is equivalent to disabling the feedback system, since new values are simply discarded: the first value to get in the feedback loop is retained forever. Plugging the feedback directly, without any filtering ($\alpha = 1$), leads to high instability, with the system continuously oscillating. Among the other values analysed, $\alpha = 0.01$ shows a smooth behaviour, with an impact on the system comparable to $\alpha = 0.1$.

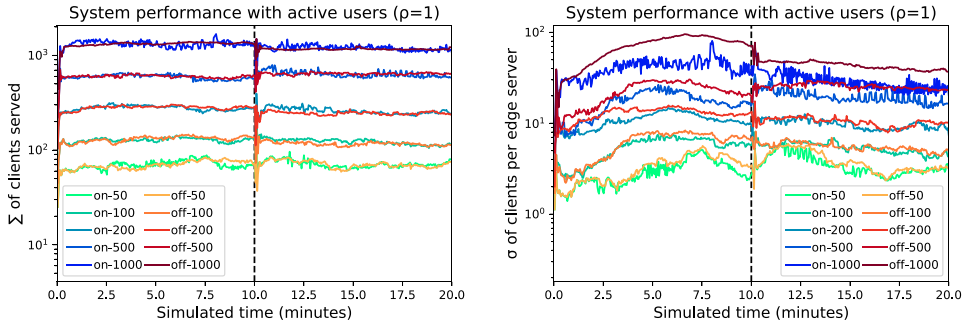


Fig. 6. System correctness. Warm colours show the system behaviour with no feedback, cold colours with the feedback system enabled and $\alpha = 10^{-2}$. Both configurations are able to cope with the system dynamics, serving all the users, and actually slightly "overserve" them. This is due to a combination of network propagation and elaboration times and absence of an explicit handoff procedure for leaving a region, resulting in users joining a different region having their streams counted also in the region they left for a few seconds. The feedback system provides benefits in terms of load balancing, as depicted in the right chart: the lower σ means lower disparity among leaders in the number of served users. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 3
Measures for the first case study.

Name	Description	Unit
E of feedback adjustment	Mean of the feedback adjustment for every leader. It measures how much the radius of the coordinated region is extended. Lower values indicate larger regions.	m
σ of feedback adjustment	Standard deviation of the feedback adjustment for every leader. It is an indication of how much the radius of the coordinated region varies among leaders. Higher values indicate higher disparity in such values, meaning that the feedback system is altering the region sizes more intensively.	m
$\sum$ of clients per edge server	Overall number of users being served. The ideal value is the number of users in the system. Higher values indicate streams being processed by multiple leaders (due to users crossing the region boundaries), lower values imply that some users are not being served.	users
σ of clients per edge server	Standard deviation of number of users served by each leader. Indication of load balancing. Higher values indicate that more computational capacity is required for some leaders w.r.t. others. The lower, the better balanced is the load.	users

of gracefully terminating the session within a region, which is unsolvable in the general case, as e.g., a device may turn off due to a depleted battery, then get on again in a different area when the battery is replaced, hence being unable to execute a graceful termination procedure.

Finally, resilience of the system to failures is analysed by observing the response to sudden disruptions hitting the leaders. Data in Fig. 7 indicates that SCR reaches stability quickly after the disruption, even when it involves a large fraction of previously elected leaders, and regardless of whether the feedback system is implemented. At disruption time, the system enters an inconsistent state, with several nodes not served and several others overserved, as they participate multiple, quickly changing regions, with their streams getting lost because of the time required to recover both regions and spanning trees for data accumulation. The feedback system does not show a measurable

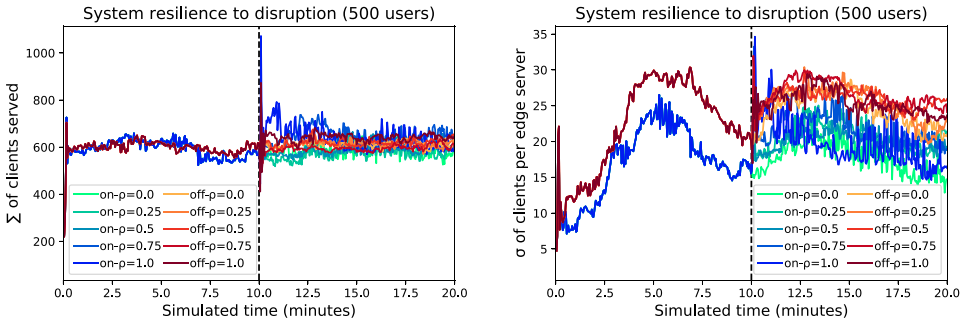


Fig. 7. System resilience to disruption. Both pattern configurations provide resilience: the system is able to find new leaders shortly after a disruptive event, even when none of the previously selected leaders remains available. The system with feedback achieves slightly better performance for smaller disruptions, but shows slower stabilisation times in the worst case. As seen in Fig. 6, the feedbacked system features visibly improved performance in terms of load balancing, both before and after the disruptive event, and regardless the degree of disruption.

Table 4
Hardware specifications for the simulated compute network.

Type	CPU	Logic cores	MIPS[a]
System on chip	Raspberry PI 2	4	1686[b]
Edge server	AMD Ryzen 7 1700X	16	38037
HPC	AMD EPYC 7401P	48	80190

[a]As computed by the 7-zip compression tool benchmark, data from OpenBenchmarking.org: http://archive.ph/UmNw1.
[b]Data unavailable on OpenBenchmarking.org at writing time and obtained from: http://archive.ph/PPTtH.

impact on resilience, however it improves load balancing (by allowing regions to dynamically resize) both before and after disruption event.

5.4. Dynamic device cluster formation in hierarchical networks

As a second scenario, consider a grid networked system, where devices are more and more powerful towards the network core. End users are located towards the edge of such network, equipped with devices that are computationally weak relatively to the network core. Users generate computationally intensive tasks, possibly more than their own device can promptly deal with: in such case, they want to offload the task to another network device. This kind of hybrid task allocation strategy (between purely local execution and full offloading to some server) requires a pre-existing or dynamic form of integration between devices, and then a collective agreement on which device should be in charge of executing the task at hand. A good task allocation system should balance the load across the infrastructure, considering the current load for each network node, and trying to minimise both latency (i.e., allocate the task as close as possible to the requester) as well as resource consumption. The goal, in this scenario, is to dynamically create clusters of devices, in which devices with less resources can offload tasks to those suffering less computational pressure. To address the important issue of "mastering continuous change", this integration of devices should be self-organising, with no supervisor node, resilient to disconnection of devices and able to automatically integrate new members. Moreover, clusters need to be able to form both vertically (i.e., among devices located along a path towards the network core) and horizontally (i.e., among devices that compose the same logical layer in the network).

5.4.1. Experimental configuration

We simulate a high performance computing infrastructure composed of a three-layered hierarchical network of devices progressively more powerful towards the core of the network. Specifically, the system features (i) 100 low power, system-on-chip (SoC) devices at the edge of the network, (ii) 25 desktop PC-level edge servers on the intermediate layer, and (iii) 10 high performance computers (HPCs) at the network core. Device technical specifications are provided in Table 4, where computational power is expressed as millions of instructions per second (MIPS). Devices are connected by reliable links, we do not simulate packet loss due to network instability and assume a reliable transport protocol (e.g., TCP/IP). Fig. 8 depicts the system, showing the network topology as well.

All devices but HPCs generate tasks following a Poisson arrival process with frequency λ. We stress the compute network by simulating a peak load of five minutes:

$$\lambda = \begin{cases} 10^{-3}\ \text{Hz} & 100\ \text{s} \leq t \leq 400\ \text{s} \\ \lambda_P\ \text{Hz} & \text{otherwise} \end{cases}$$

where t is the current time, and λ_P is the peak task frequency. Every task has a size s, measured in millions of instructions. In case a task gets generated on a device already waiting for the results of other 20 tasks to be completed, the task gets dropped. This measure is in place to keep high λ_P values from crashing the system by generating more tasks than the infrastructure could support.

The network is programmed with a SCR-based task allocation algorithm written in Protelis and executed with a round frequency of 1 Hz, working as follows: devices can measure how much free available computational capacity they have left (P_f) by counting how many cores are currently unoccupied. This value is used as input of a low pass filter, an exponential backoff function, to obtain the current perceived capacity:

$$P_p(t) = \begin{cases} P_f & t \leq 0\ \text{s} \\ \alpha P_f(t) + (1-\alpha) P_p(t-1) & \text{otherwise} \end{cases}$$

As in the previous example, this allows some degree of control over sudden changes of P_f. Devices use P_p as a distance estimator, computed only among neighbouring nodes: the distance between two nodes is the minimum between their values of P_p. The idea behind this choice is that the less computational capacity

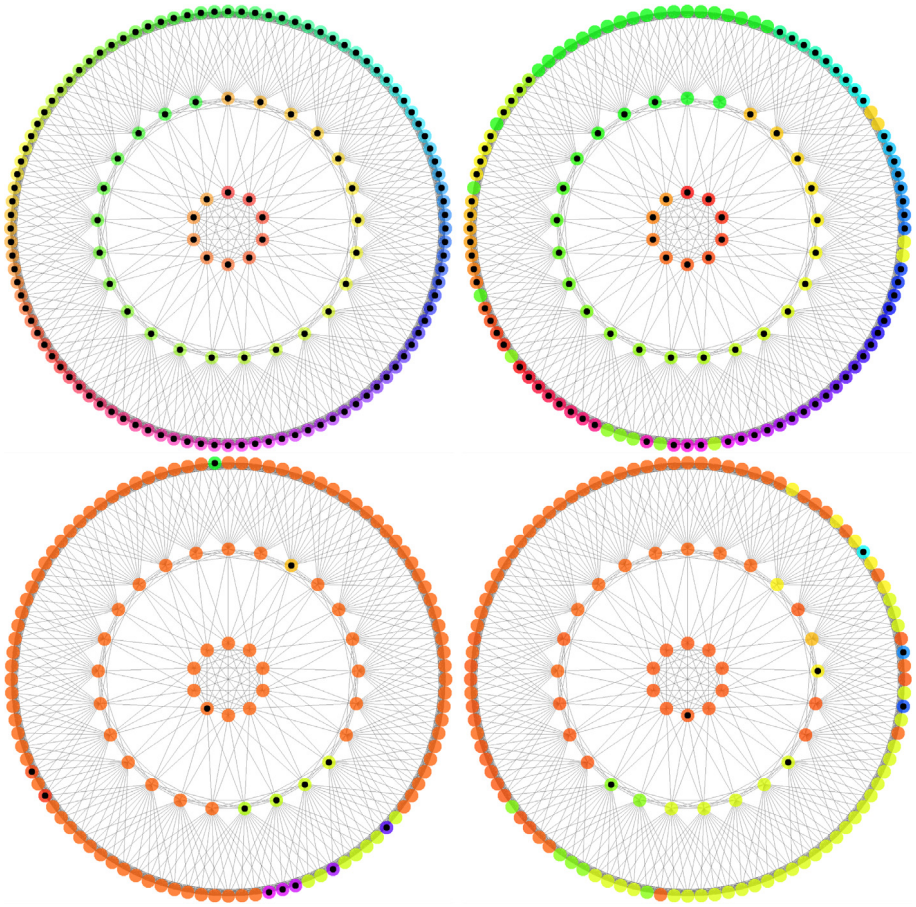


Fig. 8. Sequence of snapshots of the second case study. Nodes have progressively more computational resources towards the core of the network (inner circle). Leader nodes are identified by a black dot. Same colour corresponds to same cluster. Colours are assigned progressively to leaders by changing hue, hence clusters led by close leaders may appear similar in colour—yet of course any partition has always a single leader. Initially (top left), under mild load, every device is able to provide for itself: each one is a leader, and deals with its own tasks. Raising the task arrival frequency causes devices to form clusters (top right), getting logically closer and self-integrating to more computationally free nodes. If the pressure is higher than the overall infrastructure capacity, clusters may become very large (bottom left). Inside clusters, the leading node changes depending on the current load: the less busy and more powerful unit is the one accepting the next job. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

is available, the more likely a device is to ask for help, i.e., to form a cluster with others and offload the task to a leader. This peculiar distance metric is measured in MIPS; it is unrelated with the actual device locality, and generates a non-Euclidean space: in fact, the direct path between two nodes with high capacity left could be longer than a path transiting through a third device with very low capacity. Devices form clusters with all devices within some distance g, where g is a parameter of the self-integration algorithm, and must be chosen considering the actual devices composing the network. Although techniques for estimating a "good" value for a network could be deployed (e.g., a method could be setting it to the minimum of the maximum capacity value among all devices, while more refined methods could also consider the cores count), in this case study we test for a number of pre-defined values. Overall, in this scenario we have two non self-tuning parameters: α, with the exact same meaning and for the same reasons of the parameter with the same name described in Section 5.3.1; and g, which we assign a value statically, but that could in principle get computed based on system's metrics.

Inside a cluster, a single device is elected, based on an election biased to favour devices with higher P_p. When a task is generated inside a cluster, it is sent upstream to the leader, and, once completed, the result is sent back downstream. The task allocation to a leader causes a reduction on its P_f, and hence on its P_p: when too many tasks have been allocated, the leadership will switch to another device inside the cluster with higher P_p.

The process of self-integration into clusters is depicted in Fig. 8. We execute simulations starting at $t = 0$ s and ending at $t = 600$ s, varying g, α, λ_p, and s—these free variables are summarised in Table 5. Values of g are selected in a range that encompasses values close to the computational capacity of the weakest node. Values of 1600 and below are slightly lower, and imply that weak devices will try to accomplish some work before asking for help; while 1700 is slightly higher, thus making the weakest devices always seek for help before even trying to solve a task themselves. Very high values for g would lead the system to aggregate to a single cluster, selecting a single leader at a time, and switching leader when its load status makes it worse than another device. For each simulation run, we measure the number of task completed with success (T_s) and dropped (T_d) throughout the whole experiment, the number of tasks currently awaiting completion (T_w), the number of clusters (A), and the mean of P_p and P_f across all devices—metrics are summarised in Table 6.

5.4.2. Results

As first step, we use the first simulation results to tune the g and α parameters. The effect of varying g is depicted in Fig. 9. Despite the value guaranteeing better performance is the highest in the tested range, we did not want to forcibly prevent weaker devices from operating in solitude. Since performance are very similar among the other tested value, we choose $g = 1500$.

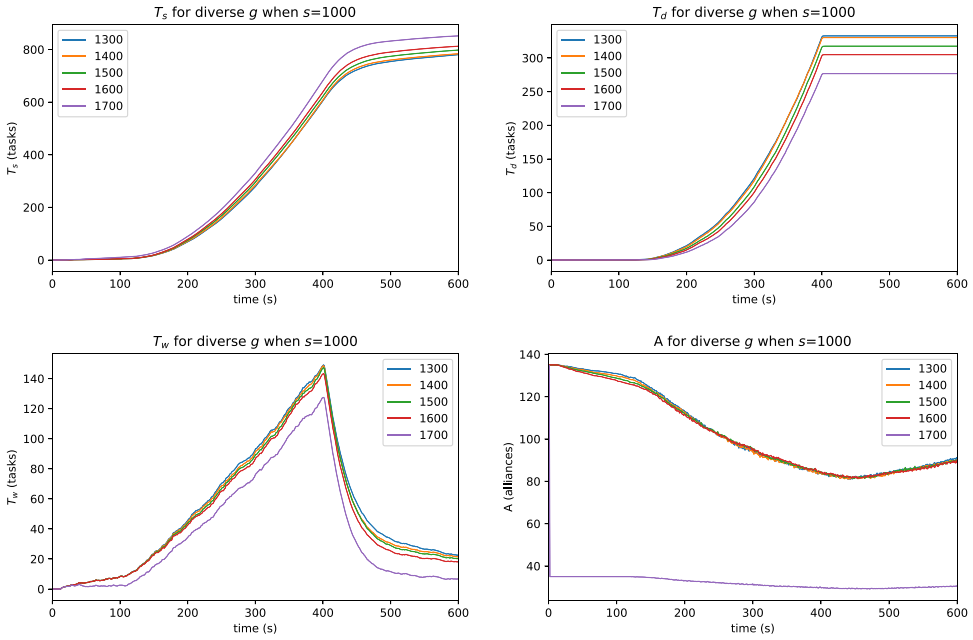


Fig. 9. Effect of diverse values for the clustering threshold parameter g. Under the conditions we used for tuning g ($s = 1000$ tasks), larger clusters improve performance (top left) and reduce the number of failing tasks (top right). However, values larger than the maximum computational capacity of peripheral nodes make them always part of a larger cluster (bottom right). In this specific case, the devices at the network core are so much faster than those populating the outermost layer that involving them in computations immediately allows for much shorter task completion time (bottom left), and, consequently, better performance.

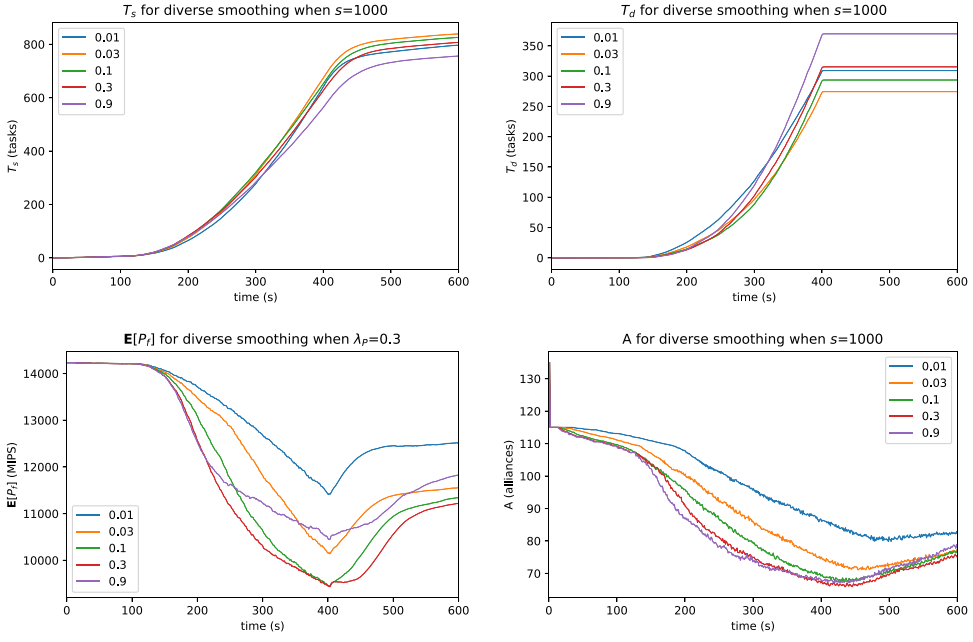


Fig. 10. Effect of diverse values for the exponential backoff parameter α. The overall system performance initially improves for larger α values, but then decreases for values greater than 0.1. The effect is noticeable by looking at the count of tasks successfully completed (top left) and dropped (top right). This happens despite the count of dynamic clusters dropping faster with a more reactive (bigger) α, and hence their size getting bigger (which usually increases performance as discussed for Fig. 9). However, there is no performance increase, and the reason is clarified by the data depicted on the bottom left, showing the average unused computational capacity of the system with a fixed task arrival peak rate $\lambda_P = 0.3$ Hz: despite few, large clusters, the power does not get exploited efficiently. This is usually a sign of the system reconfiguring the integration too often, not providing enough time for the new configuration to try to face the task peak load. Different α values obtain similar performance: this provides evidence of the robustness of SCR (and of the proposed implementation) to changes in control parameters.

Next, we need to balance the reactivity of the system in order to achieve fast adaptation while preventing oscillatory or resonant behaviours, namely, we need to better understand the effect of modifying α. Data framed in Fig. 10 show two interesting effects: first, performance reaches an optimum for α values in the [0.03, 0.1] range, then drops; higher values allow for larger clusters, but they do not improve performance as they did when analysing the system behaviour for diverse values of g, because the computational power of the system is left unexploited. This phenomenon is most likely caused by the system self-configuring

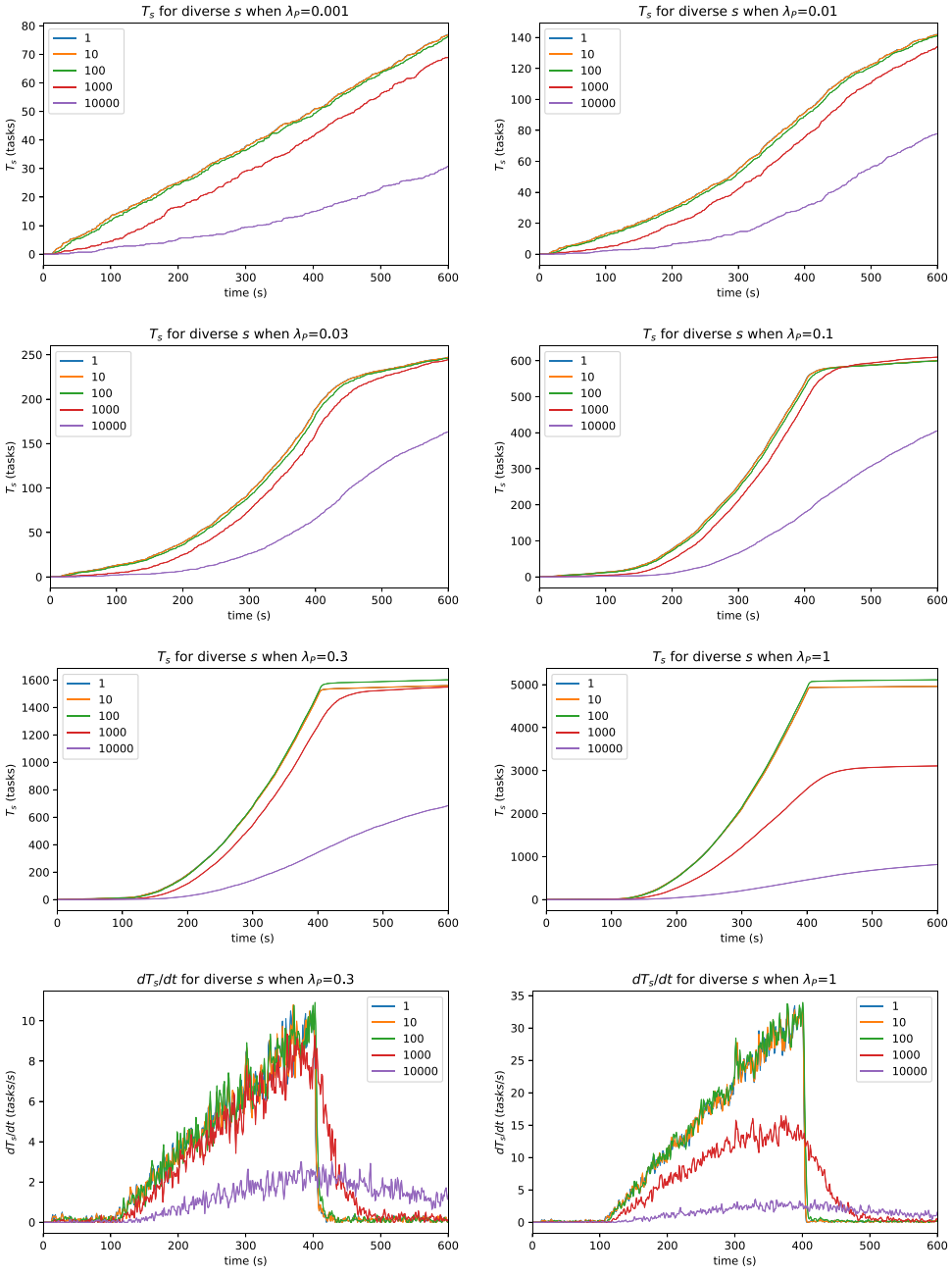


Fig. 11. Top six charts depict successfully completed tasks for different task arrival peak rates (one per chart, increasing left to right and top to bottom) and different task sizes (coloured lines). Bottom two charts frame the ex-post differentiation over time for the charts immediately above them. The self-aggregation system is able to cope with the traffic peak, and responds by progressively, dynamically increasing its ability to successfully complete tasks with time. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 5
Free variables for the second scenario.

Name	Description	Values
g	Distance threshold under which a device considers itself clustered with a neighbour	[1300, 1400, 1500, 1600, 1700] MIPS
α	Backoff algorithm parameter	[0.01, 0.03, 0.1, 0.3, 0.9]
λ_p	Task arrival peak frequency	[10^{-3}, 0.003, 10^{-2}, 0.03, 0.1, 0.3, 1] Hz
s	Task size	[1, 10, 10^2, 10^3, 10^4] millions of instructions

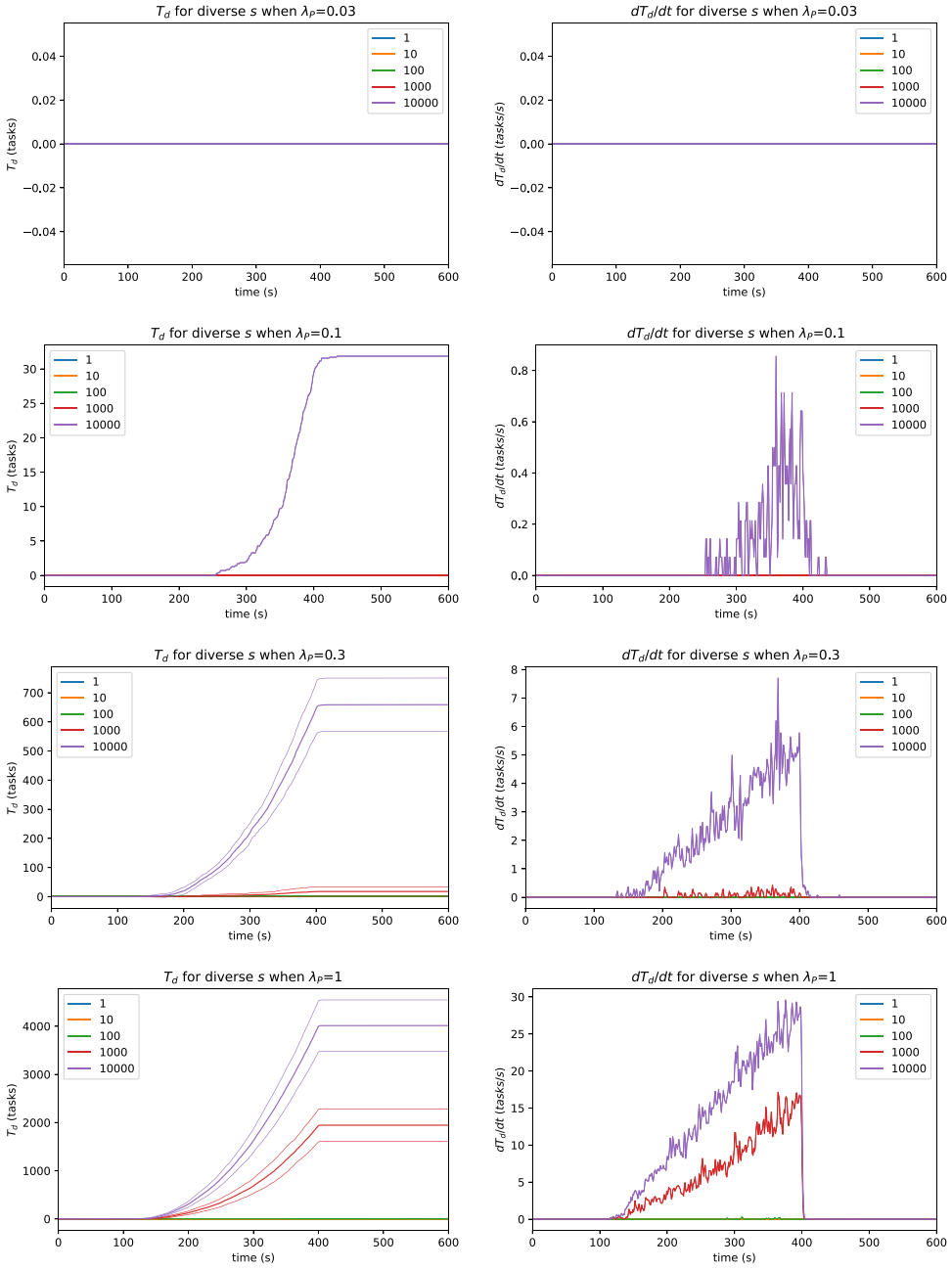


Fig. 12. Dropped tasks analysis. Left column shows the overall number of dropped tasks for increasing peak task arrival frequency (from top to bottom) and task sizes (coloured lines). Thinner lines, where present, show value ± standard deviation. Right column depicts, for each chart on the left hand side, the ex-post differentiation over time. Data shows how the system adapts to serve higher loads, and how only in case the requests actually overtake the overall available capacity it begins to drop part of the submitted tasks. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 6
Measures for the second case study. Derivatives are computed ex-post, namely, simulations generate data for T_s and T_d, and these data is then processed to compute its derivative.

Name	Description	Unit
$\mathbf{E}[P_p]$	Mean across devices of the perceived available computation capacity.	MIPS
$\mathbf{E}[P_f]$	Mean across devices of the available computation capacity.	MIPS
T_s	Overall number of tasks completed successfully throughout the experiment	tasks
$^{dT_s}/_{dt}$	Derivative of T_s	$^{\text{tasks}}/_{s}$
T_d	Overall number of tasks dropped successfully throughout the experiment	tasks
$^{dT_d}/_{dt}$	Derivative of T_d	$^{\text{tasks}}/_{s}$
T_w	Overall number of tasks awaiting completion	tasks
A	Clusters count	clusters

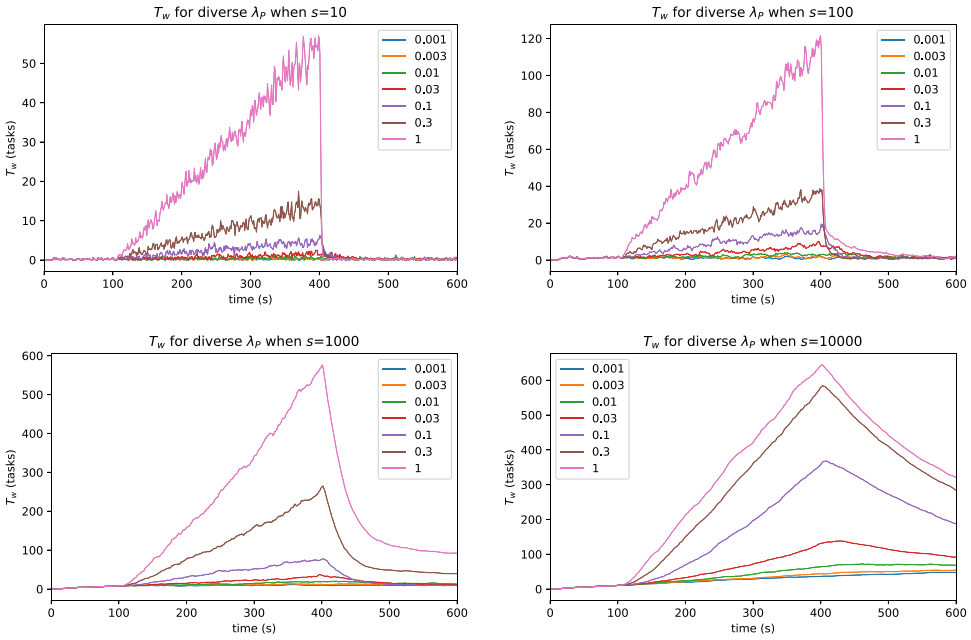


Fig. 13. Waiting queue evolution for increasingly high task sizes (from left to right and top to bottom). Different colours represent different peak task arrival frequencies. The system scales linearly. Episodes of sub-linear scaling that appear for greater task sizes and higher frequencies are mostly due to dropped tasks (see Fig. 12). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

the integration too often, not providing enough time for the new configuration to try to establish up- and down-stream and try to face the task peak load. Despite 0.03 achieving the best performance overall, we decided to favour higher responsiveness and used 0.1, which achieves similar results.

Once parameters are set, we can measure the algorithm performance in detail, with the goal of showing how the SCR pattern can easily support self-integration of subsystems. Fig. 11 analyses the evolution of the number of successfully completed tasks. Data show that the system automatically reconfigures, creating clusters of devices and splitting load within them. Derivatives show that the system is able to increasingly improve its response to higher load. The complementary side of the analysis on task completion performance is summarised in Fig. 12, which shows how the system gives up tasks. First, it is always able to successfully complete almost every tasks for s up to 100 MIPS for any λ_P and for any s if $\lambda_P \leq 0.03$. Beyond these thresholds, the system gracefully begins to drop tasks. As the differential charts show, the drop rate grows sublinearly, confirming the progressively improving response to change of the SCR-based self-integration algorithm. Finally, in Fig. 13 the count of enqueued tasks per node is reported. Data is consistent with the performance and the after-peak recovery times depicted in the previous analysis.

6. Conclusion

In this paper, we introduce *Self-organising Coordination Regions*, a pattern enabling self-integration via hybrid coordination, especially suitable to for dynamic, opportunistic scenarios where control and decision making cannot be completely centralised nor fully decentralised. The pattern fits a problem of growing relevance in a fashion particularly well suited for edge systems and for deploying a coordination stance that covers more than pure locality yet without requiring any global coordinator. In particular, it helps to tackle complexity and integration challenges (introduced in Section 1) in large-scale, dynamic systems amenable to hybrid control. It does so by combining the divide-and-conquer principle with a self-organisation process (dynamically sustaining a set of leader-regulated regions and the corresponding inter- and intra-interactions) that promotes self-improving integration. In particular: centralisation of decision making responsibilities to a small set of leaders reduces contention and the dispersal of non-local monitoring and control (such that most of the components can operate on a local basis); the regional structure provides tunable scopes for collaboration and the problem domain; and the feedback loops foster self-* activities.

To show applicability and benefits, we also present two case studies in situated edge and hierarchical network computing, showing that the pattern is able to create self-improving structures of semi-independent coordination regions where feedback-based interactions regulate activity. The pattern is also easily extensible: we show, e.g., how a simple feedback mechanism could be devised to improve the load balancing across different leaders. We believe the presented pattern, along with the proposed implementation leveraging the Aggregate Computing framework and its library of reusable self-stabilising building blocks can streamline the prototyping and development of a wide class of advanced coordination mechanisms, especially in the context of Edge Computing.

The contribution of this paper can be framed in the autonomic computing (generally) and SISSY (specifically) research areas. By synthesising a significant corpus of recurrent architectural problem/solution pairs into a recognisable and well-documented pattern, it fosters design of adaptive cooperation for large-scale, dynamic ecosystems in continuous operation. Moreover, we note that, as happened with the pattern described in this paper, aggregate computing captures a level of abstraction that very well fits with the problem of turning "complex machinery" into coherent "building blocks". This can favour the emergence of new design patterns for distributed self-organising systems, which we expect to identify in the contexts of complex situation recognition, distributed learning, and distributed tracking of complex phenomena.

Specifically, as short-term future work, we would like to investigate variants of the pattern, improve its analysis by a control-theoretical perspective, and possibly follow research directions

along the SISSY challenge, for instance those related to learning and computational trust (e.g., on the line explored in [75,76]).

CRediT authorship contribution statement

Danilo Pianini: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing - original draft, Writing - review & editing, Visualization. **Roberto Casadei:** Conceptualization, Methodology, Software, Validation, Writing - original draft, Writing - review & editing, Visualization. **Mirko Viroli:** Conceptualization, Methodology, Writing - original draft, Resources, Writing - review & editing, Funding acquisition. **Antonio Natali:** Conceptualization, Methodology, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] J.O. Kephart, D.M. Chess, The vision of autonomic computing, IEEE Comput. 36 (1) (2003) 41–50, http://dx.doi.org/10.1109/MC.2003.1160055.

[2] K.L. Bellman, J. Botev, A. Diaconescu, L. Esterle, C. Gruhl, C. Landauer, P.R. Lewis, A. Stein, S. Tomforde, R.P. Würtz, Self-improving system integration - status and challenges after five years of SISSY, in: 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems, FAS*W, Trento, Italy, September 3-7, 2018, IEEE, 2018, pp. 160–167, http://dx.doi.org/10.1109/FAS-W.2018.00042, [105].

[3] M.W. Maier, Architecting principles for systems-of-systems, Syst. Eng. 1 (4) (1998) 267–284, http://dx.doi.org/10.1002/(sici)1520-6858(1998)1:4<267::aid-sys3>3.0.co;2-d.

[4] S. Tomforde, J. Hähner, H. Seebach, W. Reif, B. Sick, A. Wacker, I. Scholtes, Engineering and mastering interwoven systems, in: W. Stechele, T. Wild (Eds.), ARCS 2014 - 27th International Conference on Architecture of Computing Systems, Workshop Proceedings, February 25–28, 2014, Luebeck, Germany, University of Luebeck, Institute of Computer Engineering, VDE Verlag / IEEE Xplore, 2014, pp. 1–8, URL http://ieeexplore.ieee.org/document/6775093/.

[5] C. Müller-Schloer, H. Schmeck, T. Ungerer (Eds.), Organic Computing - A Paradigm Shift for Complex Systems, Springer, 2011, http://dx.doi.org/10.1007/978-3-0348-0130-0.

[6] A. Diaconescu, L.J.D. Felice, P. Mellodge, Multi-scale feedbacks for large-scale coordination in self-systems, in: 13th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2019, Umea, Sweden, June 16–20, 2019, IEEE, 2019, pp. 137–142, http://dx.doi.org/10.1109/SASO.2019.00025.

[7] R. Casadei, D. Pianini, M. Viroli, A. Natali, Self-organising coordination regions: A pattern for edge computing, 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, in: Lecture Notes in Computer Science, vol. 11533, Springer, 2019, pp. 182–199, http://dx.doi.org/10.1007/978-3-030-22397-7_11, [106].

[8] D. Weyns, B.R. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, K.M. Göschka, On patterns for decentralized control in self-adaptive systems, in: R. de Lemos, H. Giese, H.A. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24–29, 2010 Revised Selected and Invited Papers, in: Lecture Notes in Computer Science, vol. 7475, Springer, 2010, pp. 76–107, http://dx.doi.org/10.1007/978-3-642-35813-5_4.

[9] W. Jaradat, A. Dearle, A. Barker, Towards an autonomous decentralized orchestration system, Concurr. Comput.: Pract. Exper. 28 (11) (2016) 3164–3179, http://dx.doi.org/10.1002/cpe.3655.

[10] 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, IEEE, 2019, pp. 60–67, http://dx.doi.org/10.1109/FMEC.2019.8795355.

[11] J. Beal, D. Pianini, M. Viroli, Aggregate programming for the internet of things, IEEE Comput. 48 (9) (2015) 22–30, http://dx.doi.org/10.1109/MC.2015.261.

[12] R. Casadei, G. Fortino, D. Pianini, W. Russo, C. Savaglio, M. Viroli, A development approach for collective opportunistic Edge-of-Things services, Inform. Sci. 498 (2019) 154–169, http://dx.doi.org/10.1016/j.ins.2019.05.058.

[13] P.M. Walker, S.A. Amraii, N. Chakraborty, M. Lewis, K.P. Sycara, Human control of robot swarms with dynamic leaders, in: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14–18, 2014, IEEE, 2014, pp. 1108–1113, http://dx.doi.org/10.1109/IROS.2014.6942696.

[14] M. Díaz, B. Rubio, J.M. Troya, A coordination middleware for wireless sensor networks, in: Systems Communications 2005 (ICW / ICHSN / ICMCS / SENET 2005), 14–17 August 2005, Montreal, Canada, IEEE Computer Society, 2005, pp. 377–382, http://dx.doi.org/10.1109/ICW.2005.5.

[15] R. de Cássia Acioli Lima, N.S. Rosa, I.R.L. Marques, TS-mid: Middleware for wireless sensor networks based on tuple space, in: 22nd International Conference on Advanced Information Networking and Applications, AINA 2008, Workshops Proceedings, GinoWan, Okinawa, Japan, March 25–28, 2008, IEEE Computer Society, 2008, pp. 886–891, http://dx.doi.org/10.1109/WAINA.2008.244.

[16] J. Liu, J. Liu, J. Reich, P. Cheung, F. Zhao, Distributed group management in sensor networks: Algorithms and applications to localization and tracking, Telecommun. Syst. 26 (2–4) (2004) 235–251, http://dx.doi.org/10.1023/B:TELS.0000029041.37854.92.

[17] D. Pianini, S. Dobson, M. Viroli, Self-stabilising target counting in wireless sensor networks using euler integration, in: 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA, September 18–22, 2017, IEEE Computer Society, 2017, pp. 11–20, http://dx.doi.org/10.1109/SASO.2017.10, [107].

[18] E. Gamma, R. Helm, R. Johnson, J.M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, first ed., Addison-Wesley Professional, 1994.

[19] G. Coulouris, J. Dollimore, T. Kindberg, Distributed Systems - Concepts and Designs, third ed., in: International Computer Science Series, Addison-Wesley-Longman, 2002.

[20] V. Lesch, C. Krupitzer, S. Tomforde, Emerging self-integration through coordination of autonomous adaptive systems, in: FAS*W@SASO/ICAC, IEEE, 2019, pp. 6–9.

[21] K.L. Bellman, C. Gruhl, C. Landauer, S. Tomforde, Self-improving system integration - On a definition and characteristics of the challenge, in: FAS*W@SASO/ICAC, IEEE, 2019, pp. 1–3.

[22] J. Branke, M. Mnif, C. Müller-Schloer, H. Prothmann, U. Richter, F. Rochner, H. Schmeck, Organic computing - Addressing complexity by controlled self-organization, in: ISoLA, IEEE Computer Society, 2006, pp. 185–191.

[23] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, Wiley, 1996.

[24] C. Alexander, S. Ishikawa, M. Silverstein, A Pattern Language: Towns, Buildings, Construction, Oxford University Press, 1977, URL http://www.amazon.fr/exec/obidos/ASIN/0195019199/citeulike04-21.

[25] J.M. Smith, Elemental Design Patterns (Paperback), Addison-Wesley, 2012.

[26] T. Parr, Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages, first ed., Pragmatic Bookshelf, 2009.

[27] D.C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects, Wiley, 2000, URL https://www.safaribooksonline.com/library/view/pattern-oriented-software-architecture/9781118725177/.

[28] G. Hohpe, B. Woolf, Enterprise Integration Patterns, in: The Addison-Wesley Signature Series, Prentice Hall, 2004, URL http://books.google.com.au/books?id=dH9zp14-1KYC.

[29] V. Vernon, Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka, first ed., Addison-Wesley Professional, 2015.

[30] R. Kuhn, B. Hanafee, J. Allen, Reactive Design Patterns, Manning, 2017, URL https://books.google.it/books?id=tYPAsgEACAAJ.

[31] M. Casciaro, Node.js Design Patterns, second ed., in: Community Experience Distilled, Packt, 2016, URL https://books.google.it/books?id=Ys4GBgAAQBAJ.

[32] R. Hanmer, Patterns for Fault Tolerant Software, Wiley Publishing, 2007.

[33] S. Hayden, C. Carrick, Q. Yang, et al., Architectural design patterns for multiagent coordination, in: 3rd Int. Conf. on Agent Systems, Vol. 99, 1999, p..

[34] B. Horling, V.R. Lesser, A survey of multi-agent organizational paradigms, Knowl. Eng. Rev. 19 (4) (2004) 281–316, http://dx.doi.org/10.1017/S0269888905000317.

[35] J.L. Fernandez-Marquez, G.D.M. Serugendo, S. Montagna, M. Viroli, J.L. Arcos, Description and composition of bio-inspired design patterns: a complete overview, Nat. Comput. 12 (1) (2013) 43–67, http://dx.doi.org/10.1007/s11047-012-9324-y.

[36] O. Babaoglu, G. Canright, A. Deutsch, G.A.D. Caro, F. Ducatelle, L.M. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, A. Montresor, et al., Design patterns from biology for distributed computing, ACM Trans. Auton. Adapt. Syst. (TAAS) 1 (1) (2006) 26–66.

[37] M. Viroli, G. Audrito, J. Beal, F. Damiani, D. Pianini, Engineering resilient collective adaptive systems by self-stabilisation, ACM Trans. Model. Comput. Simul. 28 (2) (2018) 16:1–16:28, http://dx.doi.org/10.1145/3177774.

[38] T.D. Wolf, T. Holvoet, Design patterns for decentralised coordination in self-organising emergent systems, in: S. Brueckner, S. Hassas, M. Jelasity, D. Yamins (Eds.), Engineering Self-Organising Systems, 4th International Workshop, ESOA 2006, Hakodate, Japan, May 9, 2006, Revised and Invited Papers, in: Lecture Notes in Computer Science, vol. 4335, Springer, 2006, pp. 28–49, http://dx.doi.org/10.1007/978-3-540-69868-5_3.

[39] S. Frey, A. Diaconescu, I. Demeure, Architectural integration patterns for autonomic management systems, in: 9th IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems, EASe 2012, Novi Sad, Serbia, 2012, URL https://hal.telecom-paris.fr/hal-02286263.

[40] M. Magnaudet, S. Chatty, What should adaptivity mean to interactive software programmers?, in: F. Paternò, C. Santoro, J. Ziegler (Eds.), ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'14, Rome, Italy, June 17–20, 2014, ACM, 2014, pp. 13–22, http://dx.doi.org/10.1145/2607023.2607028.

[41] T.D. Wolf, T. Holvoet, Designing self-organising emergent systems based on information flows and feedback-loops, in: Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, Boston, MA, USA, July 9–11, 2007, IEEE Computer Society, 2007, pp. 295–298, http://dx.doi.org/10.1109/SASO.2007.16.

[42] 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, IEEE Computer Society, 2017, pp. 91–100, http://dx.doi.org/10.1109/SASO.2017.18, [107].

[43] A. Lluch-Lafuente, M. Loreti, U. Montanari, Asynchronous distributed execution of fixpoint-based computational fields, Log. Methods Comput. Sci. 13 (1) (2017) http://dx.doi.org/10.23638/LMCS-13(1:13)2017.

[44] S. Dasgupta, J. Beal, A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm, in: 55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12–14, 2016, IEEE, 2016, pp. 7282–7287, http://dx.doi.org/10.1109/CDC.2016.7799393.

[45] J. Beal, J. Bachrach, D. Vickery, M.M. Tobenkin, Fast self-healing gradients, in: Proceedings of the 2008 ACM Symposium on Applied Computing, SAC, Fortaleza, Ceara, Brazil, March 16–20, 2008, 2008, pp. 1969–1975, http://dx.doi.org/10.1145/1363686.1364163.

[46] F. Li, Y. Ding, K. Hao, A dynamic leader-follower strategy for multi-robot systems, in: 2015 IEEE International Conference on Systems, Man, and Cybernetics, Kowloon Tong, Hong Kong, October 9–12, 2015, IEEE, 2015, pp. 298–303, http://dx.doi.org/10.1109/SMC.2015.64.

[47] W. Ren, R.W. Beard, Distributed Consensus in Multi-vehicle Cooperative Control - Theory and Applications, in: Communications and Control Engineering, Springer, 2008, http://dx.doi.org/10.1007/978-1-84800-015-5.

[48] C. Yan, H. Fang, Observer-based distributed leader-follower tracking control: A new perspective and results, 2019, arXiv:1904.00338, CoRR abs/1904.00338, URL http://arxiv.org/abs/1904.00338.

[49] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, H. Schmeck, Towards a generic observer/controller architecture for Organic Computing, in: GI Jahrestagung (1), in: LNI, vol. P-93, GI, 2006, pp. 112–119.

[50] S. Tomforde, H. Prothmann, J. Branke, J. Hähner, M. Mnif, C. Müller-Schloer, U. Richter, H. Schmeck, Observation and control of organic systems, in: Organic Computing, Springer, 2011, pp. 325–338.

[51] P. Zahadat, Self-adaptation and self-healing behaviors via a dynamic distribution process, in: IEEE 4th International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2019, Umea, Sweden, June 16–20, 2019, IEEE, 2019, pp. 261–262, http://dx.doi.org/10.1109/FAS-W.2019.00072, [108].

[52] J. Wang, Y. Gao, K. Wang, A.K. Sangaiah, S. Lim, An affinity propagation-based self-adaptive clustering method for wireless sensor networks, Sensors 19 (11) (2019) 2579.

[53] S. Raghuwanshi, A. Mishra, A self-adaptive clustering based algorithm for increased energy-efficiency and scalability in wireless sensor networks, in: 2003 IEEE 58th Vehicular Technology Conference. VTC 2003-Fall (IEEE Cat. No. 03CH37484), Vol. 5, IEEE, 2003, pp. 2921–2925.

[54] E. Sakhaee, K. Leibnitz, N. Wakamiya, M. Murata, Bio-inspired layered clustering scheme for self-adaptive control in wireless sensor networks, in: 2009 2nd International Symposium on Applied Sciences in Biomedical and Communication Technologies, IEEE, 2009, pp. 1–6.

[55] 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, http://dx.doi.org/10.1109/SCC.2019.00019.

[56] C. Zhang, V.R. Lesser, S. Abdallah, Self-organization for coordinating decentralized reinforcement learning, in: W. van der Hoek, G.A. Kaminka, Y. Lespérance, M. Luck, S. Sen (Eds.), 9th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2010, Toronto, Canada, May 10–14, 2010, Volume 1-3, IFAAMAS, 2010, pp. 739–746, URL https://dl.acm.org/citation.cfm?id=1838304.

[57] R.L. Stewart, R.A. Russell, A distributed feedback mechanism to regulate wall construction by a robotic swarm, Adapt. Behav. 14 (1) (2006) 21–51, http://dx.doi.org/10.1177/105971230601400104.

[58] C. Intanagonwiwat, R. Govindan, D. Estrin, J.S. Heidemann, F. Silva, Directed diffusion for wireless sensor networking, IEEE/ACM Trans. Netw. 11 (1) (2003) 2–16, http://dx.doi.org/10.1109/TNET.2002.808417.

[59] 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, URL http://www.usenix.org/events/nsdi04/tech/welsh.html.

[60] L. Mottola, G.P. Picco, Logical neighborhoods: A programming abstraction for wireless sensor networks, in: DCOSS, in: Lecture Notes in Computer Science, vol. 4026, Springer, 2006, pp. 150–168.

[61] Y.A. Alrahman, R. De Nicola, M. Loreti, Programming interactions in collective adaptive systems by relying on attribute-based communication, Sci. Comput. Program. 192 (2020) 102428.

[62] J. Beal, S. Dulman, K. Usbeck, M. Viroli, N. Correll, Organizing the aggregate: Languages for spatial computing, 2012, arXiv:1202.5509, CoRR abs/1202.5509, URL http://arxiv.org/abs/1202.5509.

[63] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, D. Pianini, From distributed coordination to field calculus and aggregate computing, J. Log. Algebraic Methods Program. 109 (2019).

[64] L. He, P. Bai, X. Liang, J. Zhang, W. Wang, Feedback formation control of UAV swarm with multiple implicit leaders, Aerosp. Sci. Technol. 72 (2018) 327–334, http://dx.doi.org/10.1016/j.ast.2017.11.020.

[65] R. Haghighi, C. Cheah, Multi-group coordination control for robot swarms, Automatica 48 (10) (2012) 2526–2534, http://dx.doi.org/10.1016/j.automatica.2012.03.028.

[66] J. Jin, X. Ma, Hierarchical multi-agent control of traffic lights based on collective learning, Eng. Appl. AI 68 (2018) 236–248, http://dx.doi.org/10.1016/j.engappai.2017.10.013.

[67] O. Yadgar, S. Kraus, C.L.O. Jr., Hierarchical information combination in large-scale multiagent resource management, in: M. Huget (Ed.), Communication in Multiagent Systems, Agent Communication Languages and Conversation Polocies, in: Lecture Notes in Computer Science, vol. 2650, Springer, 2003, pp. 129–145, http://dx.doi.org/10.1007/978-3-540-44972-0_6.

[68] A. Paulos, S. Dasgupta, J. Beal, Y. Mo, K.D. Hoang, L.J. Bryan, P.P. Pal, R.E. Schantz, J. Schewe, R. Sitaraman, A. Wald, C. Wayllace, W. Yeoh, A framework for self-adaptive dispersal of computing services, in: IEEE 4th International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2019, Umea, Sweden, June 16–20, 2019, IEEE, 2019, pp. 98–103, http://dx.doi.org/10.1109/FAS-W.2019.00036, [108].

[69] 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, in: Lecture Notes in Computer Science, vol. 10789, Springer, 2018, pp. 94–122, http://dx.doi.org/10.1007/978-3-030-00302-9_4.

[70] 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, in: Lecture Notes in Computer Science, vol. 11533, Springer, 2019, pp. 200–217, http://dx.doi.org/10.1007/978-3-030-22397-7_12, [106].

[71] A. Shimbel, Structure in communication nets, in: Proceedings of the Symposium on Information Networks (New York, 1954), Polytechnic Press of the Polytechnic Institute of Brooklyn, 1955, pp. 199–203.

[72] R. Bellman, On a routing problem, Quart. Appl. Math. 16 (1) (1958) 87–90, http://dx.doi.org/10.1090/qam/102435.

[73] S.D. Stoller, Leader election in asynchronous distributed systems, IEEE Trans. Comput. 49 (3) (2000) 283–284, http://dx.doi.org/10.1109/12.841132.

[74] Y. Mo, J. Beal, S. Dasgupta, An aggregate computing approach to self-stabilizing leader election, in: 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems, FAS*W, Trento, Italy, September 3–7, 2018, IEEE, 2018, pp. 112–117, http://dx.doi.org/10.1109/FAS-W.2018.00034, [105].

[75] R. Casadei, A. Aldini, M. Viroli, Towards attack-resistant Aggregate Computing using trust mechanisms, Sci. Comput. Program. 167 (2018) 114–137.

[76] S. Edenhofer, S. Tomforde, J. Kantert, L. Klejnowski, Y. Bernard, J. Hähner, C. Müller-Schloer, Trust communities: An open, self-organised social infrastructure of autonomous agents, in: Trustworthy Open Self-Organising Systems, in: Autonomic Systems, Springer, 2016, pp. 127–152.

[77] G. Cabri, N. Capodieci, L. Cesari, R. De Nicola, R. Pugliese, F. Tiezzi, F. Zambonelli, Self-expression and dynamic attribute-based ensembles in SCEL, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8–11, 2014, Proceedings, Part I, in: Lecture Notes in Computer Science, vol. 8802, Springer, 2014, pp. 147–163, http://dx.doi.org/10.1007/978-3-662-45234-9_11.

[78] 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, http://dx.doi.org/10.1109/FAS-W.2017.116, [109].

[79] K. Birman, The promise, and limitations, of gossip protocols, Oper. Syst. Rev. 41 (5) (2007) 8–13, http://dx.doi.org/10.1145/1317379.1317382.

[80] D. Pianini, J. Beal, M. Viroli, Improving gossip dynamics through overlapping replicates, in: A. Lluch-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, in: Lecture Notes in Computer Science, vol. 9686, Springer, 2016, pp. 192–207, http://dx.doi.org/10.1007/978-3-319-39519-7_12.

[81] G. Audrito, F. Damiani, M. Viroli, Optimal single-path information propagation in gradient-based algorithms, Sci. Comput. Program. 166 (2018) 146–166, http://dx.doi.org/10.1016/j.scico.2018.06.002.

[82] Y. Mo, J. Beal, S. Dasgupta, Error in self-stabilizing spanning-tree estimation of collective state, 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. 1–6, http://dx.doi.org/10.1109/FAS-W.2017.112, [109].

[83] G. Audrito, S. Bergamini, F. Damiani, M. Viroli, Effective collective summarisation of distributed data in mobile multi-agent systems, in: E. Elkind, M. Veloso, N. Agmon, M.E. Taylor (Eds.), Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13–17, 2019, International Foundation for Autonomous Agents and Multiagent Systems, 2019, pp. 1618–1626, URL http://dl.acm.org/citation.cfm?id=3331882.

[84] J. Hannemann, G. Kiczales, Design pattern implementation in Java and aspectJ, in: Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4–8, 2002, 2002, pp. 161–173, http://dx.doi.org/10.1145/582419.582436.

[85] M. Odersky, T. Rompf, Unifying functional and object-oriented programming with Scala, Commun. ACM 57 (4) (2014) 76–86, http://dx.doi.org/10.1145/2591013.

[86] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, DEECO: An ensemble-based component system, in: Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering, in: CBSE '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 81–90, http://dx.doi.org/10.1145/2465449.2465462.

[87] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, D. Pianini, From field-based coordination to aggregate computing, in: G.D.M. Serugendo, M. Loreti (Eds.), Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Held As Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18–21, 2018. Proceedings, in: Lecture Notes in Computer Science, vol. 10852, Springer, 2018, pp. 252–279, http://dx.doi.org/10.1007/978-3-319-92408-3_12.

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

[89] D. Pianini, M. Viroli, J. Beal, Protelis: practical aggregate programming, in: R.L. Wainwright, J.M. Corchado, A. Bechini, J. Hong (Eds.), Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13–17, 2015, ACM, 2015, pp. 1846–1853, http://dx.doi.org/10.1145/2695664.2695913.

[90] M. Viroli, R. Casadei, D. Pianini, Simulating large-scale aggregate MASs with alchemist and scala, in: M. Ganzha, L.A. Maciaszek, M. Paprzycki (Eds.), Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, GdaŃSk, Poland, September 11–14, 2016, in: Annals of Computer Science and Information Systems, vol. 8, IEEE, 2016, pp. 1495–1504, http://dx.doi.org/10.15439/2016F407.

[91] J. Beal, M. Viroli, Building blocks for aggregate programming of self-organising applications, in: Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014, London, United Kingdom, September 8–12, 2014, 2014, pp. 8–13, http://dx.doi.org/10.1109/SASOW.2014.6.

[92] F. Bonomi, R.A. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: M. Gerla, D. Huang (Eds.), Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC@SIGCOMM 2012, Helsinki, Finland, August 17, 2012, ACM, 2012, pp. 13–16, http://dx.doi.org/10.1145/2342509.2342513.

[93] L.M.V. González, L. Rodero-Merino, Finding your way in the fog: Towards a comprehensive definition of fog computing, Comput. Commun. Rev. 44 (5) (2014) 27–32, http://dx.doi.org/10.1145/2677046.2677052.

[94] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: Vision and challenges, IEEE Internet Things J. 3 (5) (2016) 637–646, http://dx.doi.org/10.1109/JIOT.2016.2579198.

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

[96] S. Montagna, D. Pianini, M. Viroli, A model for drosophila melanogaster development from a single cell to stripe pattern formation, in: Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26–30, 2012, 2012, pp. 1406–1412, http://dx.doi.org/10.1145/2245276.2231999, https://doi.org/10.1145/2245276.2231999.

[97] V.D. Florio, M. Bakhouya, A. Coronato, G.D. Marzo, Models and concepts for socio-technical complex systems: Towards fractal social organizations, Syst. Res. Behav. Sci. 30 (6) (2013) 750–772, http://dx.doi.org/10.1002/sres.2242.

[98] 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, http://dx.doi.org/10.1016/j.future.2018.09.005.

[99] S. Hoyer, J. Hamman, Xarray: N-D labeled arrays and datasets in Python, J. Open Res. Softw. 5 (1) (2017) http://dx.doi.org/10.5334/jors.148.

[100] J.D. Hunter, Matplotlib: A 2D graphics environment, Comput. Sci. Eng. 9 (3) (2007) 90–95, http://dx.doi.org/10.1109/MCSE.2007.55.

[101] K. Bilal, A. Erbad, Edge computing for interactive media and video streaming, in: Second International Conference on Fog and Mobile Edge Computing, FMEC 2017, Valencia, Spain, May 8–11, 2017, IEEE, 2017, pp. 68–73, http://dx.doi.org/10.1109/FMEC.2017.7946410.

[102] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, A. Puliafito, R. Buyya, Metropolitan intelligent surveillance systems for urban areas by harnessing IoT and edge computing paradigms, Softw., Pract. Exper. 48 (8) (2018) 1475–1492, http://dx.doi.org/10.1002/spe.2586.

[103] M. de Sá, E.F. Churchill, Mobile augmented reality: A design perspective, in: Human Factors in Augmented Reality Environments, Springer, 2012, pp. 139–164, http://dx.doi.org/10.1007/978-1-4614-4205-9_6.

[104] M.M. Haklay, P. Weber, Openstreetmap: User-generated street maps, IEEE Pervas. Comput. 7 (4) (2008) 12–18, http://dx.doi.org/10.1109/MPRV.2008.80.

[105] 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems, FAS*W, Trento, Italy, September 3–7, 2018, IEEE, 2018, URL https://ieeexplore.ieee.org/xpl/conhome/8598461/proceeding.

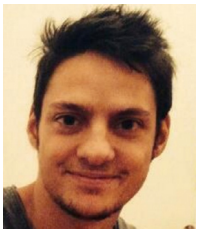
[106] H.R. Nielson, E. Tuosto (Eds.), Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings, in: Lecture Notes in Computer Science, vol. 11533, Springer, 2019, http://dx.doi.org/10.1007/978-3-030-22397-7.

[107] 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA, September 18–22, 2017, IEEE Computer Society, 2017, URL https://ieeexplore.ieee.org/xpl/conhome/8063636/proceeding.

[108] IEEE 4th International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2019, Umea, Sweden, June 16–20, 2019, IEEE, 2019, URL https://ieeexplore.ieee.org/xpl/conhome/8785421/proceeding.

[109] 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, URL https://ieeexplore.ieee.org/xpl/conhome/8063634/proceeding.

Danilo Pianini is a postdoc at the Department of Computer Science and Engineering of the Alma Mater Studiorum—-Università di Bologna, Italy. He holds a Ph.D. in Computer Science Engineering, and his main research interests include simulation, (self organising) coordination, bio-inspiration, aggregate computing, pervasive systems, software engineering, and agile software development. On those subjects, he published over 50 contributes in international journals and conferences. He is the lead designer of the Alchemist simulator and the Protelis aggregate programming language.



Roberto Casadei is a Ph.D. candidate in Computer Science and Engineering at Alma Mater Studiorum—-Università di Bologna, Italy. He holds a MEng degree in Computer Science and Engineering from the same university. His research interests include software engineering, programming languages and paradigms as well as distributed and pervasive computing. Currently, his main research activities focus on self-adaptive systems, Aggregate Computing and the IoT. He leads the development of scafi, a Scala-based DSL and platform for Aggregate Computing.

Mirko Viroli took a Ph.D. in Electrical and Computer Engineering in 2002 at Università di Bologna (UNIBO), in Italy. In UNIBO, he is now Full Professor in Computer Engineering at the DISI department (Department of Computer Science and Engineering), where he is Director of Second Cycle Degree of Computer Science and Engineering, and delegate of Technological Transfer. He is an expert in foundations of computer science and programming, object-oriented programming, advanced software development, software engineering and self-adaptive/self-organising pervasive computing systems, and regularly teaches courses on these subjects. He is author of more than 250 papers, of which more than 60 on international journals. His GoogleScholar h-index is 42 with more than 6000 citations. He is member of the Editorial Board of IEEE Software magazine, and was program chair of IFIP COORDINATION 2015, the ACM Symposium on Applied Computing (SAC 2008 and 2009), and IEEE Self-Adaptive and Self-Organizing systems (SASO 2014) conferences.

Antonio Natali is Full Professor at the Department of Computer Science and Engineering (DEIS) of the Alma Mater Studiorum—-Università di Bologna. His main research interests are in programming languages, software architectures, software engineering and eLearning. He is co-author of more than one hundred articles, papers and three books and participated to several European (Esprit and Multi Annual Projects), Italian, and regional projects on logistics, open source software, and eLearning. He has been the scientific tutor of more than ten doctorate students.