Future Generation Computer Systems 158 (2024) 11–27

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

A general framework and decentralised algorithms for collective computational processes

Giorgio Audrito [a], Roberto Casadei [b,*], Gianluca Torta [a]

[a] Università di Torino, Italy
[b] Alma Mater Studiorum—Università di Bologna, Italy

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.8310743

Keywords:
Collective adaptive systems
Collective computing
Dynamic ensembles
Aggregate processes
Decentralised systems
Distributed algorithms
Neighbourhood communications

ABSTRACT

Recent research on collective adaptive systems and macro-programming has shown the importance of programming abstractions for expressing the self-organising behaviour of ensembles, large and dynamic sets of collaborating devices. These generally leverage the interplay between the execution model and the program logic to steer the global-level emergent behaviour of the system. One notable example is the aggregate process abstraction: in an asynchronous round-based computational model, it allows to specify how aggregate-level computations are spawned, take form or spread on a domain of devices, and ultimately quit. Previous presentations of aggregate processes, however, are given in the formal framework of the field calculus, requiring knowledge of its syntax and articulated semantics. To provide a more accessible and language-agnostic presentation of such an abstraction, in this paper we introduce a general formal framework of collective computational processes (CCP). Specifically, as key contribution, we model and describe the programming interface (`spawn` construct) and dynamics of CCPs on event structures. Furthermore, we also propose novel algorithms for efficient propagation and termination of CCPs, based on statistics on the information speed and a notion of progressive wave-like closure. Crucially, thanks to our theoretical framework, we can provide optimality guarantees for the proposed algorithms, whose performance, superior to the state of the art, is assessed by simulation. Finally, to show applicability of CCPs, we provide a case study of situated service discovery in peer-to-peer networks.

1. Introduction

The Internet of Things (IoT) [1] and related trends suggest that an important class of systems involves those featuring large numbers of situated devices that self-organise to provide collaborative services. These are also known as *collective adaptive systems (CASs)* [2,3]: groups of devices operating without a central coordinator and reacting to environment and input change coherently as a whole. By a programming perspective, there is a need for abstractions able to define, structure, scope, and dynamically manage the collective-level activities that are to be executed by such collectives of devices [2,4,5]. Abstractions and mechanisms for CASs engineering have been proposed in the context of multiple research fields such as coordination [6], multi-agent systems [7], self-organisation [8], swarm robotics [9], and *macro-programming* [5,10]. Examples of abstractions include space–time constructs [11], network-wide information flows [12], computational fields [13], and *ensembles* [8,14].

Indeed, a leitmotiv in these proposals is the definition of ways to capture *dynamic collectives*, sometimes also called *ensembles* [8,14] or *aggregates* [13]. Ensembles are groups of devices (whose members can possibly change at runtime) used to denote, e.g., providers of sensing data, executors of collective tasks, recipients for multicast communications, and so on. These ensembles have proven crucial in achieving collective and self-organising behaviours [8,14,15]. Dual to the specification of ensembles is the definition of *collective tasks* [16,17], namely tasks that are carried out collaboratively by groups of devices. Examples of tasks carried out collectively include swarm activities [18], collective movement [19], crowd sensing [17], space-aware coordination [20,21], estimation of spatio-temporal phenomena [22], creation of dynamic system structures [23,24].

Traditional language-based solutions to collective computing include supporting ensembles as first-class values [25], using *attributes* to denote groups of recipients of communication acts [26], and mechanisms for allocating and orchestrating tasks at the team level [18]. A less studied problem, first considered in the context of *field calculi* [13] with the *aggregate process* abstraction [15], revolves around *how ensembles and collective computations are related*. In this paper, we consider the

* Corresponding author.
E-mail addresses: giorgio.audrito@unito.it (G. Audrito), roby.casadei@unibo.it (R. Casadei), gianluca.torta@unito.it (G. Torta).

https://doi.org/10.1016/j.future.2024.04.020
Received 13 September 2023; Received in revised form 11 April 2024; Accepted 15 April 2024
Available online 16 April 2024

ensemble formation problem by an algorithmic perspective, assuming systems modelled as networks of devices that compute in asynchronous rounds. We take inspiration from the notion of aggregate processes and introduce a theory of *collective computational process (CCP)*, modelling concurrent collective tasks running on dynamic domains of interconnected devices. On this, we thoroughly study how dynamic ensembles and CCPs could form, evolve, and cease to exist. We focus on a programming perspective, proposing algorithms to control ensemble dynamics.

The contribution is twofold.[1] First, we provide a general formal framework, based on *augmented event structures* [29,30], for modelling dynamically evolving ensembles on asynchronous networks of neighbour-interacting devices. In this formal framework, we model CCPs. Specifically, we model *aggregate processes* [31] denotationally, without relying on specifics of *field calculi* [13], thus making their characterisation more general and accessible. Second, we propose new algorithms for controlling the evolution dynamics of CCPs. This enables effective propagation and shrinking (up to extinction) across a collective, regardless of network changes and disruptions. We build our algorithms on *information speed* [32] statistics, a measure of space covered by data (following connectivity structures) over time. The first algorithm exploits these to guide process extinction, while the second uses it to enact a wave-like propagation, shifting process boundaries while the process is still active. This theoretical foundation allows us to prove optimality guarantees for the proposed algorithms.

To also evaluate these techniques in practice, we simulate a message delivery scenario (paradigmatic for several applications [15]). This allows us to experimentally compare them against baseline algorithms of signal-based termination. We test the algorithms in several network configurations, and quantify the improvements in terms of success rate and efficiency (i.e. number of rounds and bandwidth), showing benefits w.r.t. solutions in previous work. Furthermore, to show applicability on more complex scenarios, we describe a simulation-based case study of situated service discovery in a peer-to-peer network. All the experimental setup has been documented and archived on Zenodo [33] for accessibility, reproducibility, and to stimulate further research on the topic.

The paper is organised as follows. Section 2 provides motivation for the work. Section 3 describes the formal framework for dynamic collectives and CCPs. Section 4 covers algorithms for controlling the lifecycle of CCPs, while providing optimality guarantees for them. Section 5 systematically evaluates the proposed techniques by simulation. Section 6 reports about the application of the proposed techniques to a more complex case study of service discovery. Section 7 presents related work. Section 8 summarises results and prospective work.

2. Motivation and high-level requirements

G. D. Abowd refers to *collective computing* as the next computing revolution after ubiquitous computing [34]. This vision is based on distributed computing and an interconnection of humans, the physical world, and computation. Accordingly, a prominent emerging viewpoint considers a large network of computing devices as a single *distributed computing platform* [13], sometimes also called a *social machine* [35]. As a machine, it can be programmed as a whole, or be given a distributed or collective task for execution—which is essentially the idea of *macro-programming* [5,10]. In this work, we consider the problem of running computational activities on such distributed machines. We call these activities *collective computational processes (CCPs)* since they (i) involve *distributed computation*; (ii) are *collective* in nature, meaning that they are carried out collaboratively by multiple devices; and (iii) are *processes*, in the sense that they involve multiple steps to complete.

We would also like such CCPs to be *scalable* and *resilient*. Thus, we focus on fully *decentralised* computations, with no central entities acting as bottlenecks or single-points-of-failure. Moreover, devices would operate *locally*: they would sense and act on a local portion of the environment (partial observability), and would interact with a limited subset of other nearby devices. Robustness is achieved through locality and replication.

We remark that these high-level requirements, namely collectiveness, progressiveness, decentralisation, scalability (e.g. to the number of devices), and resilience, are shared by multiple application domains including sensor networks [36], swarm robotics [9,15], the IoT [1], edge-fog-cloud ecosystems [37], crowds of wearable-augmented people, and the like [38].

As mentioned, CCPs are strictly related to ensembles. Examples of informal ensemble definitions include, e.g., “all the devices located in a spatial area”, “all the devices that are equipped with a temperature sensor”, “all the devices that voluntarily provide resources for computation offloading”, “all the devices that follow a given leader device”, or “all the devices that are currently needed to solve a collaborative task”. In various applications, the shape and activity of a CCP may mutually affect each other: e.g., a leader that receives a large number of requests may want to enlarge the ensemble of its workers.

In the following, we introduce a running example of situated service discovery in a pervasive ecosystem like a smart city. It is representative of the kind of collective and self-organising computations that can be expected in the aforementioned scenarios.

2.1. Example: Situated service discovery

Consider a network of devices in a smart city that may offer and request services (e.g., computing services as in volunteer computing [39]). Interactions are limited to neighbours, e.g., for scalability, latency, and privacy reasons. Service requests scan the surroundings of the requester device for offers including cost and service-level agreements (SLAs), so that the requester can choose and consume the offer that it deems best. The requester and service provider may be hops away, thus other devices need to act as relays for data and results. Then, each request could be modelled as a separate CCP: the process would spread from the requester outwards, handle the negotiation, manage the requested task execution, and provide results. In principle, this can be supported in an ad-hoc fashion, with no pre-existing infrastructure: each device must only be able to play its part in the computation and interact with neighbours.

This scenario is implemented in simulation in the case study of Section 6.

3. A general formal framework of dynamic collectives and collective tasks

In this section, we present a general formal framework for modelling dynamic collectives and collective computational processes (CCPs) running on them. This framework can be considered as a generalisation of the field-based approach of *aggregate processes* [15,31]. To facilitate reading, a summary of the notation used throughout the paper is reported in Table 1.

We present the framework in steps. First, we specify the *system model* (Section 3.1), describing the target systems we address and the main underlying assumptions, using event structures to model system executions (Section 3.1.1). Then, we provide a model of CCPs (Section 3.2), through a progressive coverage of their features, with examples.

[1] This is the extended version of a short conference paper [27] presented at the *3rd IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS'22)* [28]. In this extended version, we (i) further generalise the model, (ii) extend the evaluation with more experiments, also covering tree topologies, (iii) develop a case study of service discovery in peer-to-peer networks, (iv) provide formal optimality results, and (v) compare with various threads of related work.

3.1. Collectives and collective computations: system model

Our target systems are *collectives*, i.e., groups of largely homogeneous devices situated in some environment and interacting in a decentralised way. Examples could be swarms of robots, collections of smart devices in a smart city, or ecosystems of IoT, edge, and fog computing nodes.

Such collectives can be modelled as a dynamic graph where nodes are *devices* (possibly equipped with *sensors* and *actuators*) and edges denote *neighbouring relationships*. Only neighbour devices can directly communicate. The neighbouring relationship may be based on logical connectivity (as in an overlay network) or physical connectivity (as in actual wireless range, so that only devices sufficiently close together can directly communicate). Indirect communication may also be possible through *stigmergy*, i.e., by perceiving and affecting the environment through *sensors* and *actuators*.

We would like such systems to perform *collective computations*, namely computations that are to be carried out by multiple devices. Such computations may be long-running, require collaboration, and require dealing with the environment. Examples of collective computations include collective movement of robot groups [19], task allocation [40], self-organisation activities [41], distributed situation recognition and action [42], and so on.

Since computations are typically long-running, and need to adapt to environmental change, we let devices operate in *rounds*. In general, each round happens *asynchronously* w.r.t. the rounds of the other devices. Each round consists of the following steps.

(i) *Context acquisition step*: the device creates a snapshot of its local context by loading its previous state, sampling sensors, and retrieving the most recent message from each of its neighbours.
(ii) *Computation step*: the device evaluates the aggregate program against its context, obtaining a result that contains (a) its local output, and (b) a coordination message (*exported data*) to be broadcast to all its neighbours.
(iii) *Interaction step*: the device broadcasts the coordination message to all its neighbours and uses the output of the computation step to drive actuators.

This is a general *collective execution schema*, whose details (like frequency of rounds, retention of messages from neighbours, topology management, delivery guarantees, etc.) are left to implementations and generally depend on available infrastructure and application goals [43].

3.1.1. Augmented event structures

An overall collective execution can be modelled as an *event structure* [29], where each event denotes a round. Following the approach of [30], we enrich an event structure with further information e.g. about the device in which an event happens.

Definition 1 (*Augmented Event Structure*)**.** An *augmented event structure* is a 4-tuple $\mathbf{E} = \langle E, \rightsquigarrow, d, s\rangle$ where E is a countable set of *events*, $\rightsquigarrow \subseteq E \times E$ is a *messaging* relation, $d : E \to \Delta$ is a mapping from events to the devices where they happened, $s : E \to S$ is a mapping from events to (some representation of) sensors status, such that:

- for any device $\delta \in \Delta$, the set of events $E_\delta = \{\epsilon \in E \mid d(\epsilon) = \delta\}$ forms a sequence of chains, i.e., there are no distinct $\epsilon, \epsilon_1, \epsilon_2 \in E_\delta$ such that either $\epsilon \rightsquigarrow \epsilon_i$ for $i = 1, 2$ or $\epsilon_i \rightsquigarrow \epsilon$ for $i = 1, 2$;
- the transitive closure of $\rightsquigarrow$ forms an irreflexive partial order $< \subseteq E \times E$, called *causality relation*;
- the set $X_\epsilon = \{\epsilon' \in E \mid \epsilon' < \epsilon\} \cup \{\epsilon' \in E \mid \epsilon \rightsquigarrow \epsilon'\}$ is finite for all ϵ (i.e., $\rightsquigarrow$ and $<$ are locally finite).

Fig. 1[2] shows an example of an augmented event structure. We also introduce the following concepts and notation:

- $p(\epsilon)$ denotes the previous event at the same device, i.e., the unique $\epsilon' \in E$ such that $\epsilon' \rightsquigarrow \epsilon, d(\epsilon) = d(\epsilon')$;
- $N(\epsilon)$ denotes the *neighbours* of ϵ, i.e., the set of events $\{\epsilon' \in E \mid \epsilon' \rightsquigarrow \epsilon\}$;
- $past(\epsilon)$ denotes the set of past events for ϵ, i.e., the set of events $\{\epsilon' \in E \mid \epsilon' < \epsilon\}$;
- $past_\delta(\epsilon)$ denotes the set of past events for ϵ at the same device δ, i.e., the set $\{\epsilon' \in E \mid \epsilon' < \epsilon \wedge d(\epsilon') = d(\epsilon)\}$;
- given two events $\epsilon, \epsilon' \in E$ such that $\epsilon' \rightsquigarrow \epsilon$, their *temporal distance* $lag(\epsilon, \epsilon')$ measures how much time has passed in ϵ since the interaction with ϵ' happened;
- given two events $\epsilon, \epsilon' \in E$ such that $\epsilon' \rightsquigarrow \epsilon$, their *spatial distance* $dist(\epsilon, \epsilon')$ measures how much space is covered moving from ϵ' to ϵ.

Remark 1 (*Measuring Spatio-Temporal Intervals*)**.** Estimating lag accurately can be done by keeping either a global clock, or two relative clock informations:

1. the interval in δ' between the reference time of ϵ' and the time when the message to ϵ was sent through the network interface; and
2. the interval in δ between the time when the message was received and the reference time of ϵ.

These two intervals can be added (together with an estimate of the time required for sending the message) to obtain an accurate lag.

On the other hand, estimating $dist$ accurately in practical scenarios may be more difficult. If Global Positioning System (GPS) locations are available, $dist$ could be directly computed from them, but with an uncertainty of a few meters. If GPS location is not available, or too energy intensive for the application, the receive signal strength indicator (RSSI) can also be used. This allows to obtain a reasonable estimate of distance without increasing the energy requirements of the application [45,46]. This gives more accurate results than GPS on shorter lengths, but it accumulates error over longer distances, eventually surpassing the GPS error. Furthermore, in this scenario, distances are computed *on message arrival*, which is about $lag(\epsilon, \epsilon')$ before ϵ, and during that time δ could have moved further away, increasing the actual distance. As distances will be used in minimisations to obtain the shortest paths, it is necessary to avoid systematic underestimates, that would compound in large errors over multiple hops. If the measured distance is $d(\epsilon, \epsilon')$, and the movement speed of devices can be bounded by v, we can use the overestimate:

$$dist(\epsilon, \epsilon') = d(\epsilon, \epsilon') + v \cdot lag(\epsilon, \epsilon').$$

If the precision obtained through GPS or RSSI is not acceptable for the application at hand, the most common alternative is using *message round-trip time* to compute a distance estimate with an uncertainty in the order of the centimetre. In this setting, measurements have to be scheduled with a limited frequency in order to avoid saturating the communication medium and raising excessively the energy requirements. A similar correction of the estimated distance using the speed bound v and the lag since the last measurement has to be used also in this context.

Example 1 (*Gradient*)**.** A gradient [32] is a fundamental building block of self-organising behaviours [47]. It denotes a class of decentralised algorithms for computing everywhere in a network of devices, the minimum distance from each device to its closest *source* device. We suppose that the information about whether a device is a source or not is local to each device—e.g., it may be read through a local sensor. One of its simplest versions, the *Adaptive Bellman–Ford (ABF)* algorithm [48], only assumes that each device is able to get (an estimation of) its distance to each one of its neighbours. If we let S denote the set of sources, $D(\delta, \delta')$ the estimated distance between devices δ and δ', and $g(\epsilon)$ the gradient

[2] We use the Viridis colour palette [44] to improve readability.

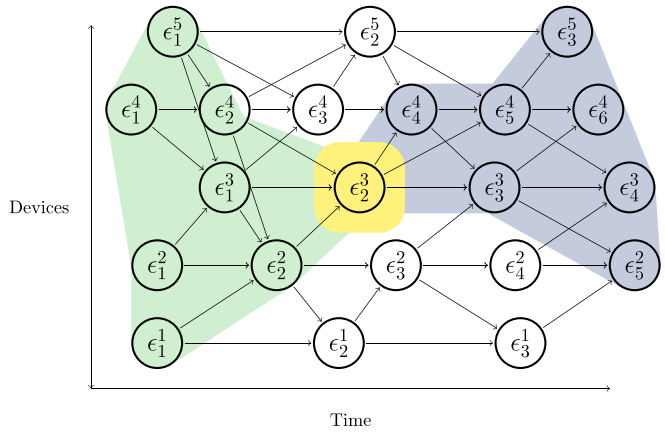

Fig. 1. Example of an event structure modelling a distributed system execution. Nodes labelled by ϵ_k^δ denote the kth round of device δ. The yellow background highlights a reference event, from which its past (green) and future (blue) are identified through the causal relationship implied by the arrows denoting neighbour events.

value computed at ϵ, then the ABF algorithm on event structures can be expressed as follows:

$$g(\epsilon) = \begin{cases} \min\limits_{\epsilon' \in N(\epsilon) \setminus p(\epsilon)} \{g(\epsilon') + D(d(\epsilon), d(\epsilon'))\} & d(\epsilon) \notin S \\ 0 & d(\epsilon) \in S \end{cases}$$

By running such function in each event, the nodes will tend to adjust their gradient value towards the "correct" distance value. Indeed, it is proved that this algorithm is self-stabilising and, more than that, globally asymptotically stable [48].

What can be done with gradients? A gradient computation induces a minimum spanning tree that can be used to diffuse or collect information hop-by-hop on shortest paths [12]. So, it is at the basis of multiple patterns of self-organising behaviour [24,47,49].

3.2. Processes over event structures

A *collective computational process (CCP)* is a transient, concurrent, computational activity carried out on a dynamic domain of devices. In other words, it is a computation that spans multiple events in an event structure. It is *transient* since it may terminate. It is *concurrent* since multiple instances of a CCP may run over the same devices at the same rounds. It is characterised by the following.

Collective processes vs. collective process instances. A *CCP process type* P is a function (computation) that expresses the behaviour and interaction to be carried out in an event (i.e., a round of a device). It can be *instantiated* to generate actual *CCP instances* P_i that will dynamically evolve accordingly to the rules introduced in this section. Any process instance P_i is assumed to be associated with a unique *process identifier (pid)* i. With no loss of generality, we assume the pid also includes *construction parameters* for a process instance, which may serve to control aspects of their behaviour.

In the following, when clear from the context, we may use the single-word term "process" to refer to either a CCP type or a CCP instance; in any case, we consider processes (types and instances) to be *collective*.

Generators and process spawning. New instances of a process P can be spawned through a *generator*. A generator G_P is a function that produces the set of identifiers $G_P(\epsilon) = \{i, j, \ldots\}$, in each event ϵ, of the process instances that need to be created, or *spawned*, in that event ϵ (which we call the *initiator* event at the *initiator* device for P_i). The idea is that any device can apply the generator in any round to get the identifiers and construction parameters of the processes to spawn.

Process execution, process output, and participation status. For each process instance P_i, we use the Boolean predicate $\pi_{P_i}(\epsilon)$ to denote whether such instance is being executed at ϵ (either being initiated by ϵ, or through propagation from previous events).

Each process instance P_i, if active in an event ϵ (i.e., $\pi_{P_i}(\epsilon) = \top$), locally computes both an *output* $O_{P_i}(\epsilon)$ (returned to the process caller) and a *status* $s_{P_i}(\epsilon)$ that can take the following values:

1. `external`: the event is not part of the process.
2. `border`: the event is at the boundary of the process.
3. `internal`: the event is in the interior part of the process.

The operation through which P_i produces its output and status is defined by its program expression (associated to the process type P). We do not cover languages for expressing process logic in this paper; the interested reader may check out *aggregate programming languages* like ScaFi [31,50] for details.

Contextual interaction and propagation of process instances to neighbours. The status of a process at a given event affects two main aspects: (i) the contextual interaction with other neighbours (e.g., a process may prescribe a device to share a locally sensed value with the device's neighbours that run the same process instance); and specifically (ii) the propagation of the process. An `external` event will not interact with other neighbours and will not propagate the process instance to neighbours. A `border` event will interact with other neighbours but will not propagate the process instance further. A `internal` event will both interact with other neighbours and propagate the process instance to them.

More formally, a process instance P_i active in an event ϵ *automatically propagates* to any event ϵ' of which ϵ is a neighbour ($\epsilon \rightsquigarrow \epsilon'$) if and only if it returns `internal` status in ϵ. In formulas:

$$\pi_{P_i}(\epsilon) = \begin{cases} \top & \text{if } i \in G_P(\epsilon) \\ \top & \text{if } \exists \epsilon' \rightsquigarrow \epsilon.\ \pi_{P_i}(\epsilon') \wedge s_{P_i}(\epsilon') = \texttt{internal} \\ \bot & \text{otherwise.} \end{cases}$$

This mechanism allows devices to dynamically enter or leave the process, which can expand or shrink in space, eventually ceasing to exist when all devices quit. For instance, a device can call itself out of a process if its hop-by-hop distance (a specific case of the *gradient* described above) from the initiator of the process exceeds a certain threshold. Although the decision of participating or not in a process instance is ultimately local, that decision may also depend on information computed collectively within the specific process instance.

An example of evolution dynamics of two concurrent processes is provided in Fig. 2.

Process result and process status/shape computation. Since a process instance computes both a status and an output for each event, a process type P can be programmed by specifying two (possibly interdependent) parts: the part computing the "shape" (i.e., the evolving domain of devices) of the process, and the part computing the output. These two parts can be equally important for the functionality, since a different domain of executing devices would provide a different set of contributions, thereby affecting the overall result. Moreover, running computations on a smaller domain of devices may also provide non-functional benefits, by using fewer resources than those needed by a larger system.

Process termination. A process is said to *have terminated* once it is not run any more by any device. It means that all the devices that previously were `internal` have switched their status to `border` or `external`—hence preventing further automatic propagation of the process to other devices. Closing a process is a matter of *coordination* among the devices that participate in the process. Various strategies can be considered to coordinate a process shutdown.

Up to this point, we have discussed the "status computation" as a very basic mechanism to regulate the production of the process

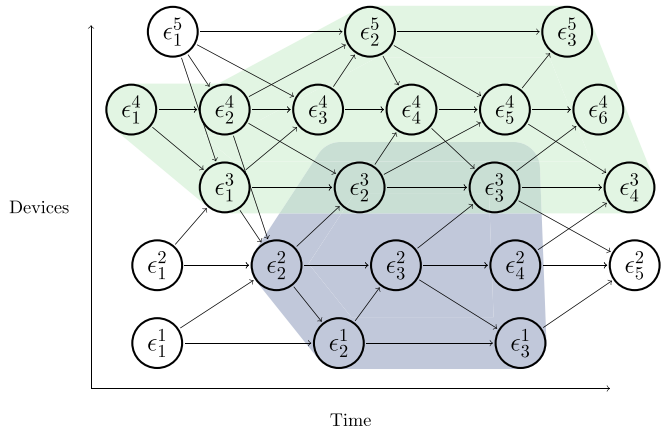

Fig. 2. Example of evolution dynamics of two collective processes. Consider a process P that propagates only within 1 hop from initiator events. Instance 1 (green) is initiated by device 4, which keeps it alive until the end of the computation, by initiating it on each one of its events ϵ_1^4 to ϵ_6^4. Instance 2 (blue) is initiated only in ϵ_2^2 and ϵ_3^2, and thus starts later and closes earlier. Notice that device δ_3 in events ϵ_2^3 and ϵ_3^3 runs two process instances at the same time. This is possible since multiple instances of the same process are allowed to overlap.

shape, i.e., to dynamically determine the set of devices that participate in a process instance. Further techniques for shape regulation and termination of processes are covered in Section 4 and are another contribution of this manuscript.

Example 2 (*Multi-Gradient*)**.** Consider the notion of a gradient, introduced in Example 1. The proposed algorithm has a problem: each device only computes the minimum distance to the closest source device. Then, how could we produce a set of values denoting the minimum distances to all source devices? This can be achieved by having a gradient process instance per source.

In particular, the CCP type corresponding to a "multi-gradient computation" could be defined as follows.

- *Process identifier.* The pid of process instances is the identifier of the source from which the gradient has to be computed.
- *Generation.* The generator function yields in source devices a singleton set with as sole element the discussed pid, and in non-source devices an empty set.
- *Computation.* The computation is the gradient computation discussed in Example 1, where the source is the device whose identifier equals the pid.
- *Status.* The kind of status computation depends on the desired process shape—e.g., whether we would like the process to be limited in space or not. In the latter case, the status is just the '`internal`' constant. In the former case, the status is '`internal`' if the gradient value computed in the process instance domain (the set of devices that run that very process instance) does not exceed a certain threshold, and '`external`' otherwise.
- *Termination.* Termination should be initiated once the source device in a process is not a source any more (e.g., because the sources are determined by a dynamic leader election strategy [51]). See Section 4 for possible strategies for organised termination.

A note on applicability. The proposed CCP abstraction can be generally applied on any system conforming to the model discussed in Section 3.1. In particular, it is suitable whenever a distributed system needs to carry out long-running tasks requiring *self-organisation* or *decentralised collaboration* among sets of devices. Most specifically, it can be applied to scenarios analogous to those addressed by similar abstractions (cf. Section 7), which include, e.g.: swarm rescue scenarios [26], multi-robot exploration [15], smart warehouse management [52], spatial coordination [21], e-vehicle fleet navigation [14], and self-organised environmental monitoring for emergency management [42].

Table 1
Summary of notation.

Symbol	Description
δ	Device identifier
ϵ	Event (round) identifier
ϵ_k^δ	The kth round of device δ
$d(\epsilon)$	Device at which event ϵ occurred
$p(\epsilon)$	Previous event at the same device
$past(\epsilon)$	Set of past events of ϵ
$past_\delta(\epsilon)$	Set of past events of ϵ on device δ
$N(\epsilon)$	Neighbour past events of event ϵ
$lag(\epsilon,\epsilon')$	Temporal distance between events
$dist(\epsilon,\epsilon')$	Spatial distance between events
P	Aggregate process
P_i	Aggregate process instance, identified by pid i
G_P	Generator producing pids $\{i\}$ for P
$\pi_{P_i}(\epsilon)$	Whether P_i is active in event ϵ
$O_{P_i}(\epsilon)$	Output of process instance P_i in ϵ
$s_{P_i}(\epsilon)$	Status of process instance P_i in ϵ
$s^*_{P_i}(\epsilon)$	Extended status returned by P_i in ϵ
$\texttt{eSpawn}_X$ ($X \in \{L,S,I,W\}$)	Extended *spawn* construct (L=legacy, S=share, I=ISPP, W=WISPP)
$TA_X(\epsilon)$	Termination awareness of event ϵ according to $spawn_X$ ($X \in \{L,S,I,W\}$)
$T_L(\epsilon)$	Termination predicate on event ϵ
$D^w(\epsilon)$ ($w \in \{dist, lag\}$)	Shortest-path distances based on weight function $w(\epsilon,\epsilon')$ and source predicate $src(\epsilon)$
$Sl(\epsilon)$	Slowness predicate on event ϵ
θ	Minimum information speed allowed

3.2.1. A construct for spawning process instances (*spawn*)

In [31], a functional language construct $spawn(P, G_P)$ was introduced to run independent instances of a process P, where new instances are locally generated according to the generator G_P. A *spawn* expression is evaluated round by round (cf. Section 3.1), and in different rounds the set provided by G_P may vary. In a round ϵ in which, e.g., $i \in G_P(\epsilon)$, a new instance of P with identifier i will be spawned locally. We remark that the computations of different instances P_i and P_j are fully independent and do not share any data. Thus, they represent distinct activities, each with its peculiar evolution and history.

The output of a $spawn(P, G_P)$ expression in an event ϵ is the set of pairs $\{(i, O_{P_i}(\epsilon)), \ldots\}$ for which predicate $\pi_{P_i}(\epsilon)$ returns true and the output status $s_{P_i}(\epsilon)$ is not `external`. In the rest of the paper, we enhance this construct and propose algorithmic techniques to improve the propagation/shrinking dynamics of processes.

Example 3 (*Service Discovery*)**.** The logic of the service discovery processes introduced in Section 2.1 can be expressed through a *spawn* (P, G_P). In this case, the G_P denotes the set of service requests to be spawned, and P is the collective computation to be run on the corresponding domains. For instance, P may denote a computation that spreads the process in space until a certain distance threshold is covered, through a gradient (cf. Example 1), and collects offers [49] by aggregating them while descending the gradient-induced spanning tree.

4. Techniques for dynamic ensembles

In this section, we describe novel techniques for dynamic ensemble formation by means of extensions to the basic *spawn* function covered in Section 3.2. The *extended spawn* `eSpawn` takes a function P similarly to *spawn*, but P can return an *extended status* that can take the additional value `terminating` indicating that the process must be quit

not only by the current node but also by all the other devices running it. In the following subsections, we shall propose four alternative versions of eSpawn that aim to achieve the best performance in terms of minimising the (computational and networking) resources required to guide process propagation and handle process termination.

4.1. Baselines

Legacy baseline ($\texttt{eSpawn}_L$*)*

First, we consider the *legacy* version of eSpawn from literature [31], denoted as $\texttt{eSpawn}_L$, and describe it in terms of how it determines the process status at each event. When a process instance P_i is active in an event ϵ (i.e., $\pi_{P_i}(\epsilon)$ is *true*), an extended status $s^*_{P_i}(\epsilon)$ is returned by P, and it is interpreted into a "regular" status $s_{P_i}(\epsilon)$ by eSpawn. Towards this aim, we define *termination awareness* for a process instance P_i and event ϵ as follows.

Definition 2. Predicate $TA_L(\epsilon)$ denotes the fact that event ϵ is *termination-aware*, i.e., either $s^*_{P_i}$ demands termination in ϵ, or some neighbour event was termination-aware since its previous round. In formulas:

$$TA_L(\epsilon) := s^*_{P_i}(\epsilon) = \texttt{terminating} \text{ or } \exists \epsilon' \in N'(\epsilon).\ TA_L(p(\epsilon')). \tag{1}$$

Above, $N'(\epsilon) = N(\epsilon) - p(\epsilon) \cup \{\epsilon\}$, that is the replacement of $p(\epsilon)$ with ϵ in $N(\epsilon)$.

According to this definition, if $TA_L(\epsilon)$ becomes *true*, it will not go back to $false$ in future events ϵ' on device $d(\epsilon)$ until $\pi_{P_i}(\epsilon')$ becomes $false$. This has the purpose of keeping track in device $d(\epsilon)$, for some rounds, that P should terminate. Actual termination occurs when both ϵ and all its neighbours $N(\epsilon)$ agree that the process should terminate, as computed by the termination predicate $T_L(\epsilon)$ defined by:

$$T_L(\epsilon) := \forall \epsilon' \in N'(\epsilon).\ TA_L(\epsilon'). \tag{2}$$

When $T_L(\epsilon)$ becomes *true*, it is time for P_i to terminate at device $d(\epsilon)$, with $s_{P_i}(\epsilon)$ taking value external (regardless of $s^*_{P_i}(\epsilon)$). If this is not the case, but $s^*_{P_i}(\epsilon) = \texttt{terminating}$, then $s_{P_i}(\epsilon) = \texttt{internal}$, indicating that ϵ is still within P_i (in order to propagate termination). In all other cases, $s_{P_i}(\epsilon) = s^*_{P_i}(\epsilon)$.

Improved baseline ($\texttt{eSpawn}_S$*)*

A significant improvement to $\texttt{eSpawn}_L$ can be easily obtained by exploiting the semantics of the *share* operator recently introduced in field calculus (FC) [53], instead of the *rep* and *nbr* operators used in $\texttt{eSpawn}_L$. In terms of computation on the event structure, the $\texttt{eSpawn}_S$ extension (subscript S is for *share*) allows a more efficient definition of predicate $TA_S(\epsilon)$ w.r.t. Eq. (1), by directly accessing the predicate in neighbour events:

$$TA_S(\epsilon) := s^*_{P_i}(\epsilon) = \texttt{terminating} \text{ or } \exists \epsilon' \in N(\epsilon).\ TA_S(\epsilon'). \tag{3}$$

The propagation of termination awareness TA_S is clearly faster in this case, since event ϵ directly exploits the values of TA_S of its neighbours ϵ', instead of that of their predecessors $p(\epsilon')$. Actual termination $T_S(\epsilon)$ in $\texttt{eSpawn}_S$ is defined as in Eq. (2), but based on the TA_S defined by Eq. (3).

4.2. Exploiting information speed

One shortcoming of $\texttt{eSpawn}_S$ is the possible resurgence of terminated processes due to some isolated nodes not receiving the termination signal. The idea for fixing it, inspired by the BIS algorithm [32], is that of estimating the spatial and temporal distances of each event ϵ from the *initiator* event ϵ_0 that has started the process. In case they correspond to an *information speed* that is below a certain threshold (more details below), it is taken as an indication of a likely disconnection from the source, prompting the device $d(\epsilon)$ to leave the process.

We call this extension $\texttt{eSpawn}_I$ (subscript I is for *Information Speed-based Process Propagation*, or *ISPP*). In $\texttt{eSpawn}_I$, termination awareness TA_I is computed as TA_S in Eq. (3). Differently from $\texttt{eSpawn}_S$, though, TA_I becoming true implies a transformation in the returned status, so that $s^*_{P_i}(\epsilon) = \texttt{internal}$ gets converted to $s_{P_i}(\epsilon) = \texttt{border}$ (cf. Eq. (5)). This implies that termination-aware nodes *do not propagate the process* to their neighbours, slowing down the process expansion and helping termination to catch up with it. Since termination-aware nodes do not propagate the process (not even to their next event), once every neighbour is termination-aware the process naturally stops. This makes a termination predicate T_I superfluous.

In addition to this difference, $\texttt{eSpawn}_I$ also features another term called $Sl(\epsilon)$ to detect a *slow* information propagation. Let us denote with $D^w(\epsilon)$ a classic shortest-path distance function based on weights $w(\epsilon, \epsilon')$ (where $\epsilon' \rightsquigarrow \epsilon$) and a *source* predicate $src(\epsilon)$. In each event, this distance estimate is updated to the smallest distance through a neighbour event, as in a step of the Bellman–Ford algorithm on the event structure:

$$D^w(\epsilon) = \min\{D^w(\epsilon') + w(\epsilon, \epsilon') :\ \epsilon' \in N(\epsilon)\}$$

starting with $D^w(\epsilon) = 0$ where $src(\epsilon)$ holds. Through D^w, an estimation of the spatial or temporal distance of events from sources can be obtained after a few rounds of computation, even for those far from sources. Assume that $src(\epsilon)$ is true in all events on the device that spawned the process by providing pid i in the generator G_P, and w is either the spatial distance $dist$ or the temporal distance lag (cf. Section 3.1.1). Assume also that devices move at a speed bounded by a given v.

A key property of D^{dist} is that, since it is a minimisation and $dist(\epsilon, p(\epsilon))$ cannot be more than $v \cdot lag(\epsilon, p(\epsilon))$ (as v is the maximum speed for a device), it always increases with a speed below v in events on a same device $d(\epsilon)$: indeed, even if the values from other neighbours raise significantly, $D^{dist}(\epsilon)$ can still keep the previous value $D^{dist}(p(\epsilon))$ incremented by $v \cdot lag(\epsilon, p(\epsilon))$. Instead, in D^{lag}, the weight $lag(\epsilon, p(\epsilon))$ is the time interval between two rounds, hence it can be shown that $D^{lag}(\epsilon)$ is always equal to the temporal distance between the current event ϵ and its most recent source event (modulo lag measurement errors). Thus, we define *slowness* as follows.

Definition 3. Predicate $Sl(\epsilon)$ (slowness) denotes the fact that the information speed detected at event ϵ is *too slow*, i.e.:

$$Sl(\epsilon) := D^{dist}(\epsilon) \leq \theta(D^{lag}(\epsilon) - \Delta_t) \tag{4}$$

where Δ_t is the average time interval between rounds, and θ is a constant representing the minimum speed of information that we are willing to allow.

When $Sl(\epsilon)$ becomes true, it causes an event to also enter the border state. Summarising, the status s_{P_i} is computed as:

$$s_{P_i}(\epsilon) := \begin{cases} \texttt{external} & \text{if } s^*_{P_i}(\epsilon) = \texttt{external} \\ \texttt{border} & \text{if } s^*_{P_i}(\epsilon) = \texttt{border} \text{ or } TA_I(\epsilon) \text{ or } Sl(\epsilon) \\ \texttt{internal} & \text{otherwise.} \end{cases} \tag{5}$$

Consider a scenario where devices move, hence possibly disconnecting from the main part of the process originated from the source. Furthermore, assume that $\theta > v$ and the actual speed of information in the network is at least θ. Events ϵ happening in the connected devices are still *not* slow, because newer events happening in the source continue to spread a 0 temporal distance that leads to $D^{lag}(\epsilon)$ to be too low for Eq. (4), since the measured speed of information will be at least θ. However, events happening on the disconnected devices no longer receive information from the source, so that their D^{lag} increases together with time while their D^{dist} increases at speed v. Since $\theta > v$, the left-hand side of Eq. (4) increases faster than the right-hand side, so that $Sl(\epsilon)$ has to eventually become *true*, stopping process propagation

and terminating that disconnected part. Without the *SI* mechanism, isolated groups of devices would continue to run the process, possibly even resurrecting it by reconnecting to parts where it had already terminated.

4.3. Wave-like propagation

The last extension of *spawn* introduced in this paper is $\texttt{eSpawn}_W$ (subscript W is for *Wave-like ISPP*). Technically, this version is similar to $\texttt{eSpawn}_I$, but the two versions exhibit fundamentally different behaviours: while $\texttt{eSpawn}_I$ aims at terminating isolated instances of the process, $\texttt{eSpawn}_W$ acts during a phase when the process is still propagating. Thus, it aims at modifying the dynamic evolution of alive processes, by terminating them in selected devices that have already acted as propagators, and are no longer required.

Specifically, the difference between $\texttt{eSpawn}_W$ and $\texttt{eSpawn}_I$ lies in the definition of the *src* predicate used by D^{dist} and D^{lag} to determine whether an event ϵ is a source event. In $\texttt{eSpawn}_W$, $src(\epsilon)$ is only true in the very first event when the process is generated by providing pid i in the generator G_P, and *not* in the following events on the same device (unlike $\texttt{eSpawn}_I$). This apparently minor difference has large consequences in the algorithm behaviour: as the source ephemerally disappears, every event behaves as if being in a disconnected part, eventually becoming *slow* and leaving the process (including the original source itself). This leads to a wave of termination that starts from the source device and propagates outwards, following the propagation of the process itself with some lag. At any given time, the process is thus active only on a set of events with similar spatial distances from the initiator ϵ_0, leading to a wave-like propagation. This allows the process to travel far through the network, while keeping its spatial extension low at all times.

4.4. Adapting to different scenarios

The newly introduced extensions both rely on a crucial parameter θ, the minimum information speed that we are willing to allow. Correctly tuning this parameter is crucial for obtaining the best performance. If θ becomes higher than the average information speed in the network, processes terminate prematurely, failing to accomplish their tasks. In order to avoid this scenario, the estimate of θ needs to be on the conservative side: however, if it is too low, the behaviour of the new extensions will degenerate to be very similar to $\texttt{eSpawn}_S$.

Furthermore, a one-fits-all number for θ is impossible, as the information speed depends on many parameters: time intervals between rounds t and their variance *tvar*, communication radius r, dimensionality of the space n, device density *dens*, movement speed *speed* and whether propagation is allowed through a single path or multiple paths. If the process is restricted to a single path, the theoretical-based estimation in [32] of *single-path* information speed can be used:

$$\theta_{sp} = \frac{8n(1+tvar^2)r}{4(n+1)t} + \frac{speed}{2}. \tag{6}$$

For instance, this could apply to the *tree topology scenario* in the experiments (cf. Section 5.2), although the peculiar way in which the spanning tree is built may practically affect such speed. If instead the propagation is allowed through multiple paths, the information speed may increase depending on the network density.

In practice, empirical estimates (obtained through simulation tools such as FCPP [54]) are the most useful. Given $n = 2$ and for a wide range of other network parameters, we found that $\theta = 2.5$ behaved reasonably well assuming that process propagation is allowed in every direction. If instead process propagation is constrained to a spanning tree, we found that $\theta = 0.3$ ensures the best results across a spectrum of network parameters.

4.5. Optimality guarantees

Before the experimental evaluation of performance that we will carry out in Section 5, we investigate some formal properties of the proposed techniques for dynamic ensembles, starting with process propagation.

Theorem 1 (*Optimal Propagation*)**.** *Assume that P is a process with an expansion criterion that does not change over time. Assume that the information speed that can be measured through lag, $dist$ in the network is always higher than the threshold θ. Then processes propagate at maximum information speed, for every spawn variant* $\texttt{eSpawn}_X$.

Proof. First, notice that the process propagation logic as described in Section 3.2 does not depend on the spawn variant of choice. Intuitively, the propagation is as fast as possible since it follows a simple broadcast strategy. More formally, consider an augmented event structure $\mathbf{E} = \langle E, \rightsquigarrow, d, s\rangle$ and an event $\epsilon \in E$ where a process instance P_i is generated: $i \in G_P(\epsilon)$. Consider the sub-network of devices that eventually should take part in the process instance P_i, which induces a substructure $\mathbf{E}'$ of events only on those devices. For any device δ', consider the first event ϵ' on δ' that is in the future of the generator event: $\epsilon < \epsilon'$. By induction on the hop-distance between ϵ and ϵ' for the various δ', we can prove that P_i is run in ϵ' (i.e., $\pi_{P_i}(\epsilon')$). The base induction holds trivially for $\delta' = d(\epsilon)$. Consider now the shortest path of events $\epsilon \rightsquigarrow \dots \rightsquigarrow \epsilon'' \rightsquigarrow \epsilon'$. By inductive hypothesis, P_i is run in the event ϵ'' that immediately precedes ϵ' (i.e., $\pi_{P_i}(\epsilon'')$). Since the expansion criterion of P does not depend on time, and $d(\epsilon'')$ is going to eventually enter the process, it must be that P_i returns an `internal` status on ϵ'', thus the process is expanded to every neighbour including ϵ'. This happens also in $\texttt{eSpawn}_I$, $\texttt{eSpawn}_W$ as the measured information speed has to be higher than θ, and thus the process state is not modified. □

Even though process propagation does not pose particular challenges, handling termination is a more delicate matter. For legacy approaches, we can only get performance bounds for networks that do not experience any disconnection.

Theorem 2 (*Legacy Termination Speed*)**.** *If a network is always connected at all times, termination propagates in* $\texttt{eSpawn}_S$ *at the maximum information speed plus a lag of one round; while in* $\texttt{eSpawn}_L$ *it propagates at a third of the information speed plus a lag of three rounds. If a network can be disconnected,* $\texttt{eSpawn}_L$ *and* $\texttt{eSpawn}_S$ *may never terminate a process instance.*

Proof. If a network can be disconnected, a node that is not supposed to terminate a process instance may enter a process, then be disconnected from the rest of the network, failing to receive the termination-awareness signal and thus never fully terminating the process. This node may also occasionally re-connect to the rest of the network, spreading the process to other devices on periodic occasions, so that the process can keep involving multiple devices for an indefinite amount of time.

Assume now instead that a network is always connected at all times. In $\texttt{eSpawn}_S$, termination-awareness propagates with a broadcast at maximum information speed. Once a device is first reached by termination awareness, in most cases it will not immediately terminate; but after a round of computation all its neighbours will become termination-aware thus allowing termination with a minimal delay. A similar procedure happens with $\texttt{eSpawn}_L$, but with information requiring an average of 1.5 rounds to travel instead of 0.5 rounds, thus requiring three times the time overall. □

In practice, temporary network disconnections are common in mobile networks. They are enough to stop the propagation of termination, making the performance of legacy approaches very poor (as we will show in Section 5). By adding the control on information speed, we

can guarantee process termination even in presence of disconnections, with a speed that depends on the difference between device movement v and the information speed bound θ.

Theorem 3 (*Termination with Information Speed*). *Assume that the information speed that can be measured through lag, $dist$ in the network is always higher than the threshold θ, and that devices move at most at speed $v < \theta$. If a network is always connected at all times, termination propagates in $\texttt{eSpawn}_I, \texttt{eSpawn}_W$ at the maximum information speed plus a lag of one round. If a network can be disconnected, termination in $\texttt{eSpawn}_I, \texttt{eSpawn}_W$ propagates with a minimum speed of $\theta - v$ plus a lag of $\frac{\theta}{\theta - v}$ rounds, from the moment when the process instance source first becomes termination-aware (for $\texttt{eSpawn}_I$), or from the moment when the process is first generated (for $\texttt{eSpawn}_W$).*

Proof. If a network is always connected at all times, the termination performance of $\texttt{eSpawn}_I, \texttt{eSpawn}_W$ matches that of $\texttt{eSpawn}_S$ as they include the same termination-awareness predicate. If a network can be disconnected, termination is still guaranteed by the slowness predicate. Once the process instance source becomes termination-aware, it stops being a source for the D^w computations, so that everywhere in the network (including disconnected parts) D^{dist} starts increasing by a speed at most v while D^{lag} starts increasing together with time. Consider a device at a spatial distance of d from the source, and assume as worst case that its initial temporal distance from the source is zero. We can then compute after how much time t Eq. (4) has to switch to false:

$$d + v \cdot t = \theta(t - \Delta_t) \Leftrightarrow d + \theta\Delta_t = (\theta - v)t \Leftrightarrow t = \frac{d + \theta\Delta_t}{\theta - v}$$

This gives us as an extra time before termination of $\frac{\theta}{\theta - v}$ rounds independent of distance, and a termination propagation speed from the instance source of $d/t = \theta - v$. □

Even though the previous result only guarantees termination for θ sufficiently greater than v, we remark that this is a very reasonable assumption for real networks. In fact, if devices move faster than how fast information can travel between them, any sort of multi-hop coordination between them becomes impossible. For $\texttt{eSpawn}_I$, for termination to start it is still required that termination-awareness first reaches the instance source. Even though there are no guarantees on the time required for it under disconnections, this is usually not a problem for two reasons: *(i)* if the instance source experiences disconnections, devices start to exit the process regardless of whether the source is termination-aware; *(ii)* even though the disconnection of *some* device is likely, the disconnection of a *specific* device is much less likely.

Finally, we can use the previous results to estimate how many devices are part of an instance in $\texttt{eSpawn}_W$.

Corollary 1 (*Wave Front Size*). *The radius of a process instance in $\texttt{eSpawn}_W$ starts from about one hop and increases at speed $\rho - \theta + v$, where ρ is the true information speed in the network.*

Proof. By Theorem 1, the outermost edge of the wave propagates with the true information speed of the network ρ, so that at instant t it has a radius of ρt. By Theorem 3 the innermost edge of the wave has a radius of $d = (\theta - v)t - \theta\Delta_t$. This gives an overall active radius of the instance of $(\rho - \theta + v)t + \theta\Delta_t$, concluding the proof. □

Although the radius of the instance always tends to increase, if $v \ll \rho$ and the estimated θ is close to the real ρ, the increase speed may be small enough to be negligible.

5. Evaluation

In this section, we evaluate the proposed algorithms through synthetic experiments. First, we present the evaluation goals (Section 5.1) and describe the simulation setup (Section 5.2). Then, we show and discuss the results for the spherical (Section 5.3.1) and tree topology (Section 5.3.2). According to modern scientific practice, the experimental framework is publicly available[3] and permanently archived on Zenodo [33], with build infrastructure and instructions, for inspection and reproducibility.

5.1. Goals

This evaluation aims to experimentally verify and analyse the algorithms proposed in this paper. In particular, we consider the following evaluations goals:

1. *Correctness.* The goal is to assess that the algorithms do provide the desired process functionality, in terms of process propagation and termination (as covered in Section 4).
2. *Performance assessment.* The goal is to compare the performance of the proposed algorithms in accomplishing the task at hand, w.r.t. the baseline $\texttt{eSpawn}_L$, in terms of convergence time and resilience.
3. *Efficiency.* The goal is to get some insight about the *cost* or overhead of the algorithms (e.g., in terms of bandwidth consumption).

5.2. Experimental setup

5.2.1. Scenarios

The basic use case implemented by all the experiments is a network of devices where, at some point in time, a source device δ_F (*from*) sends a message through a process to reach a destination device δ_T (*to*). In order to reduce the variance across runs, we fix the source to be at 25% towards the bottom left of the simulated space, and the destination to be symmetrically at 75% towards the top right of the simulated space. Note that, in the simulation, there is a global clock (unlike in a real world scenario). However, it is not available to the program run by individual nodes. Intervals between rounds are not identical (see parameter *tvar* below), and the simulator provides to each node the exact temporal distance lag between events, as well as the spatial distance $dist$ with a random Weibull-distributed error with a standard deviation of 30%. These data are used for computing the speed of information in $\texttt{eSpawn}_I$ and $\texttt{eSpawn}_W$ (see Section 4.2).

We distinguish between two scenarios based on the topology followed by communications. In the *spherical* topology, messages originate in the δ_F device and spread radially in 3D trying to reach the δ_T device. In the *tree* topology, the devices are organised in a spanning tree with a root at the middle of the simulated space, and communications follow the edges of the tree from δ_F to δ_T. In both scenarios, the events happening at the δ_T device make the process function P return status `terminating`, since the reception of the message implies the termination of the process in the whole network. For simplicity, in each run we generate only one process at time $t = 10$ and wait until time $t = 50$ for its completion. We verified that results do not change by generating several processes in each test, since processes are independent of one another.

5.2.2. Parameters and executions

We consider a large number of different executions, based on the variation of the following four parameters, that have been pointed out to define the main characteristics of a peer-to-peer network [32]:

1. *tvar* (relative variance of round intervals as percentage of the average), varying from 0% (quasi-synchronous rounds) to 40% (highly asynchronous rounds), in steps of 1%;

[3] https://github.com/fcpp-experiments/process-management

2. *dens* (average number of neighbour devices for a device), varying from 8 (very sparse network) to 28 (very dense network), in steps of 0.5;
3. *hops* (average diameter of the network in hops), varying from 4 (small networks) to 24 (relatively large networks), in steps of 0.5;
4. *speed* (movement speed as percentage of communication radius over average round interval). varying from 0% (static nodes) to 40% (highly mobile nodes), in steps of 1%.

For each scenario and setting of the parameters, we averaged the results over 1000 randomised executions, also computing their variance.

The nodes are placed at random positions in a square area whose side, as well as the number of nodes itself, is determined from the parameters above. We model movement as linear through randomly-chosen waypoints in the square. We used a simple model of connection between devices, so that communication always succeeds within a fixed radius and fails beyond it. We arbitrarily choose the spatial and temporal units so that the communication *radius* is 100 and the average round *interval* is 1. We remark that this choice does not affect the results of simulation and hence it is not useful to make it vary. The information speed threshold θ was set to 2.5 for the spherical scenario and to 0.3 for the tree scenario, as reported in Section 4.4.

5.2.3. Metrics

From each simulation run, the following metrics are extracted:

- *Delivery count* (`dcount`): the number of messages that have been delivered (i.e., the number of processes carrying a message that have reached their destination).
- *Average running processes* (`aproc`): the average number of active process instances in the network.
- *Average message size* (`asiz`): the average size (in bytes) of the messages exchanged in each round to spread and maintain processes.
- *Maximum message size* (`mmsiz`): the maximum size (in bytes) of the messages exchanged in each round to spread and maintain processes.
- *Average delay*: (`adel`) the average time taken by a message to travel from δ_F to δ_T.

We compare those metrics for each version of `eSpawn`: $\texttt{eSpawn}_L$ (*legacy*), $\texttt{eSpawn}_S$ (*share*), $\texttt{eSpawn}_I$ (*ispp*), and $\texttt{eSpawn}_W$ (*wispp*).

The metrics are connected to the goals as follows:

- `dcount` conveys how many messages are delivered, and by comparing it to the number of sent messages (which is 1, set by design of the simulated scenario) it provides empirical insights about functional correctness;
- by considering the average delay `adel` (the lower the better), it is possible to compare the relative performance of algorithms in accomplishing the task at hand (i.e., delivering a message);
- finally, `aproc`, `asiz`, and `mmsiz` provide information about the costs associated to the algorithm execution, hence supporting assessment of efficiency.

5.3. Results

5.3.1. Spherical topology

Fig. 3 (left column) shows the average number of active processes (*aproc*) for the spherical scenario, for each version of `eSpawn` varying time (row (a)), and averaged over the whole simulated time varying the four chosen parameters (rows (b)–(e)).

In resource consumption, $\texttt{eSpawn}_L$ is much worse than its extensions, while $\texttt{eSpawn}_I$ and $\texttt{eSpawn}_W$ outperform $\texttt{eSpawn}_S$, with $\texttt{eSpawn}_W$ having the lowest footprint of all. The gap between the four versions increases especially with increasing *hops*. This is expected, as the wave of terminating nodes (and information speed-driven termination) has more time to be effective in large-diameter networks.

The average size of messages exchanged resulted in the same pattern as that of *aproc*, despite $\texttt{eSpawn}_I$ and $\texttt{eSpawn}_W$ having a larger overhead per process resulting in a maximum message size of 82 bytes while $\texttt{eSpawn}_L$ and $\texttt{eSpawn}_S$ have a maximum message size of 59 bytes. All the algorithms successfully deliver the message at about the same time, and all these observations are confirmed varying *speed*, *dens*, *hops* and *tvar*. We thus omitted plots for the measures other than *aproc* as they did not convey significant additional insight.[4]

5.3.2. Tree topology

For the *tree* scenario, we implemented an adaptive algorithm computing the spanning tree based on [55]. Then, we guide process propagation by first following tree parents towards the root, and then following children that include the destination in their routing table (also computed through a literature adaptive algorithm, called *single-path collection* [56]). We considered two possible approaches for implementing the routing table: as an unordered set of children for every node of the tree, or as a 256-bit *Bloom filter* [57] to reduce the exchanged message size.

Note that the formal results of Section 4.5 still hold with this topology, assuming that the actual network is the spanning tree, i.e., ignoring the physical links between nodes that do not belong to the tree. For instance, the (estimated) distance between two nodes refers to the length of the path connecting them on the tree, instead of their distance in free space. Similarly, propagating the process by broadcast ends up in activating the process in the nodes of the tree on the path from δ_F to δ_T. An important consequence is that the tree topology is more fragile than the spherical topology to network disconnections, for it is sufficient that one of the links in the tree is temporarily lost.

Fig. 3 (right column) shows the average number of active processes (*aproc*) for the tree topology, for each version of `eSpawn` varying time (row (a)), and averaged over the whole simulated time varying the four chosen parameters (rows (b)–(e)). As expected, the percentage of nodes running the process at any given time is significantly lower than for the spherical topology (cf. Fig. 3 left column). The performance order between the four versions confirms that of the spherical scenario, with a larger margin for $\texttt{eSpawn}_W$. Also in the tree case, all algorithms successfully deliver the message with a similar and very high probability at about the same time. Only $\texttt{eSpawn}_W$ has a slightly lower probability of successful delivery, though always above 98% except for very high values of *tvar* and for middle-range values of *speed*, where the probability can drop to 70% (see Fig. 4 row (a)). Looking at that graph, it is interesting to note that the behaviour of $\texttt{eSpawn}_W$ converges to the behaviour of $\texttt{eSpawn}_I$ when the *speed* of devices reaches the value of 30% (recall that *speed* is a percentage), corresponding to the information speed threshold $\theta = 0.3$ set for the experiments with the tree topology. This is in accordance with Corollary 1, according to which when $v = \theta$ the wave only expands without contracting, increasing its radius with the true information speed ρ.

The performance in message size follows partly the values shown for *aproc*, although with a strong impact from the chosen approach for implementing the routing table, especially when varying *dens* and *hops* thus increasing the number of devices in the network. See Fig. 4 rows (b)–(e), where the left column refers to the ordered map implementation of the routing table, while the right column refers to the Bloom filter implementation. In the ordered map implementation, as for the spherical scenario, $\texttt{eSpawn}_I$ and $\texttt{eSpawn}_W$ have a larger overhead visible in the maximum message size (left column, rows (d),(e)), which is compensated by the reduction in *aproc* resulting in an average message size (left column, rows (b),(c)) that is smaller for $\texttt{eSpawn}_W$, with $\texttt{eSpawn}_I$ only slightly worse than $\texttt{eSpawn}_S$. The fixed size of the

[4] These additional figures can be accessed online at the linked repository.

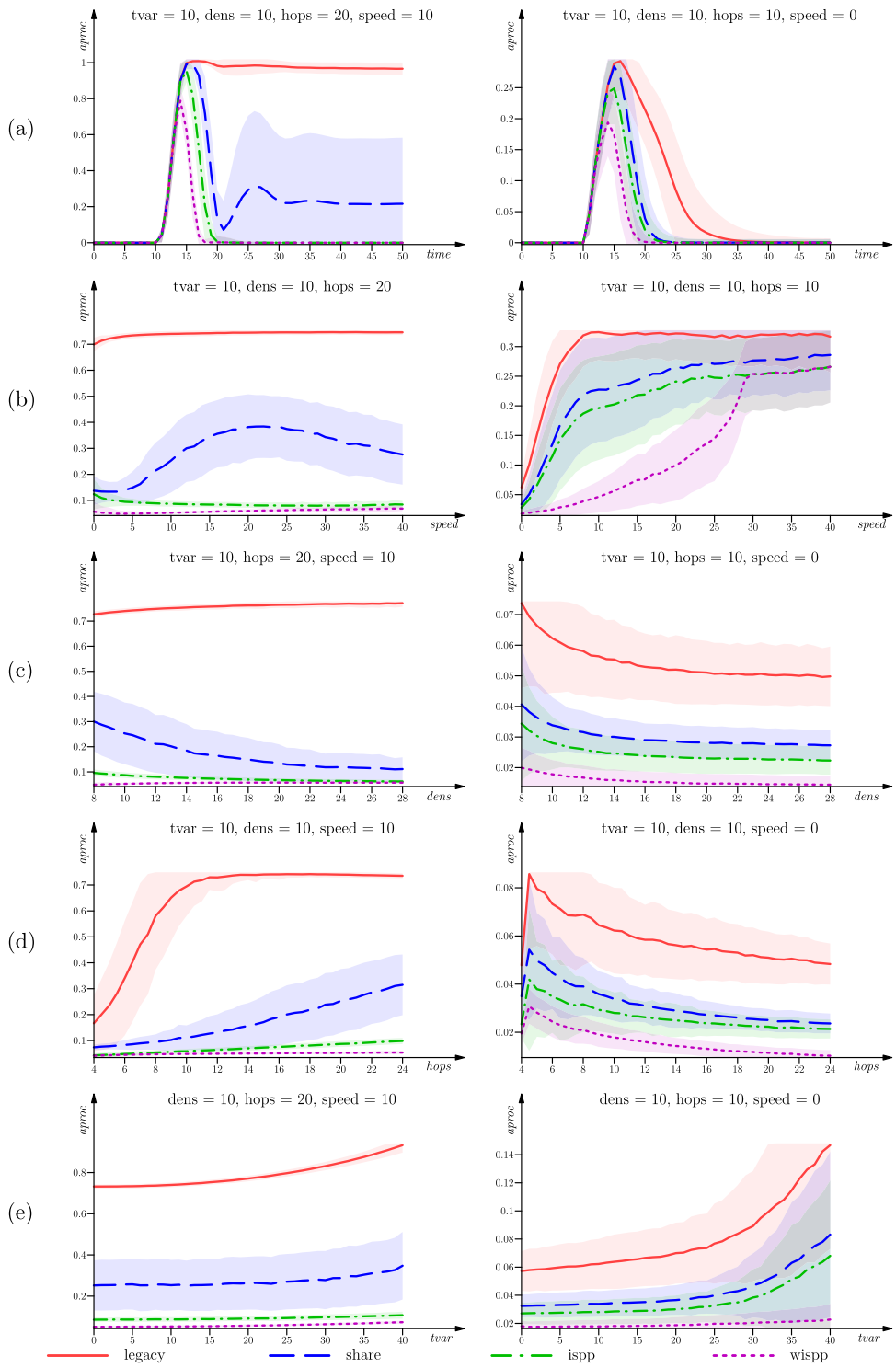

Fig. 3. Average processes (*aproc*) for the spherical (left) and tree (right) scenarios varying time, *speed*, *dens*, *hops* and *tvar*.

Bloom filter results in it performing slightly worse on average message size (right column, rows (b),(c)), except for large numbers of hops. On maximum message size (right column, rows (d),(e)), on the other hand, the performance increase is huge, improving by more than a factor of 10 in many executions.

5.3.3. Summary and key findings

It is important to stress the fact that the goal of our experiments was *not* to compare our implementation of multi-hop proximity-based message delivery with existing related techniques from the literature on computer networks (e.g. P3ON [58], ad hoc networks [59]). The notion of CCP applies to a much more general family of problems than message delivery (see Section 8.2 for a discussion about additional interesting domains).

In general, the high-level requirements we set out in Section 2 (namely, having *scalable*, *decentralised* and *collective* mechanisms to control *ensembles*), are essentially satisfied *by the very design* of the techniques studied here: the propagation and termination functions

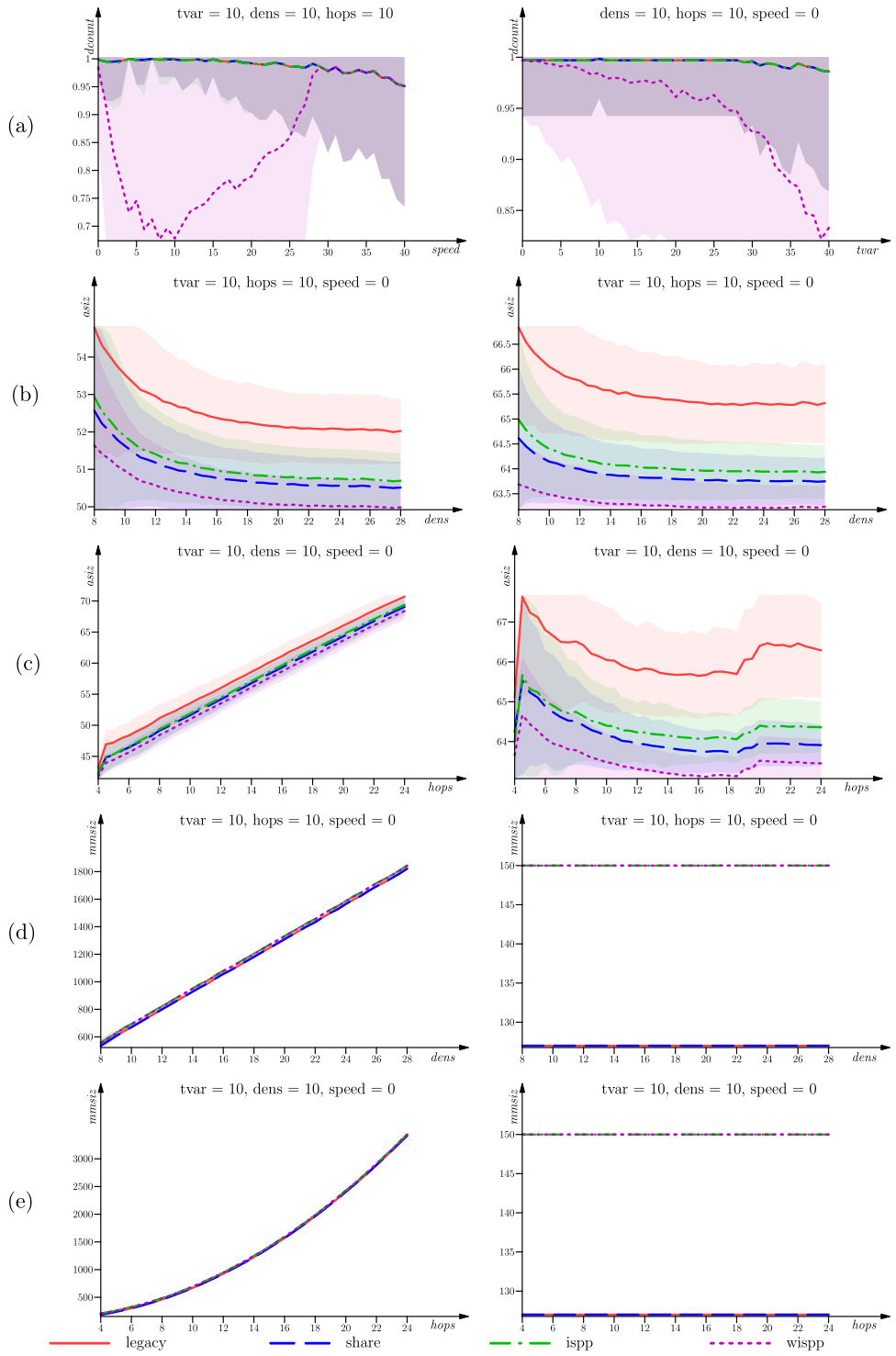

Fig. 4. Top row: delivery count as percentage in the tree scenario, varying *speed* and *tvar*. Other rows: average message size (centre) and maximum message size (bottom) using an unordered set (left column) and a Bloom filter (right column).

are indeed totally distributed, lightweight computations performed on each device, and achieve their collective global tasks by exploiting information exchange between neighbouring nodes.

Our experiments convey that the CCP abstraction provides a working solution for regulating, in a decentralised way, how an *ensemble* of devices takes shape while carrying out a collective task. In particular, we have shown that ensemble evolution can be effectively controlled with well-founded strategies, such as the ones based on information speed, that ultimately result in globally coherent local decisions of the individual devices (i.e. joining or leaving the computation). Moreover, we have elaborated how the different strategies for the evolution of CCPs can be coupled with different logical views of the underlying network (i.e., the *spherical* vs. *tree* scenarios) to exhibit complementary benefits and drawbacks (cf. efficiency and resilience).

The main challenge related to the experimental part is the estimation of the information speed parameter θ, namely the minimum information speed that we are willing to allow. This is crucial for the correct working of our algorithms. As explained in Section 4.4, an

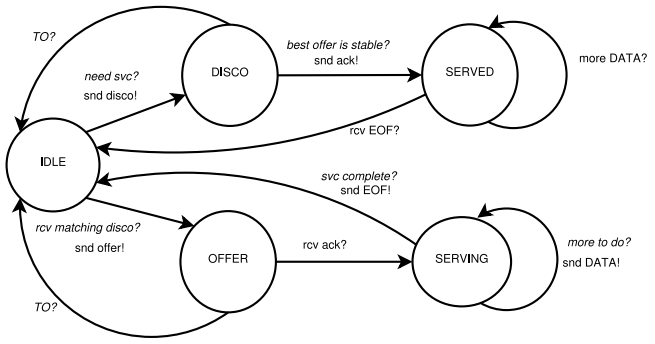

Fig. 5. Case study: the behaviour of the nodes.

effective way to deal with it is by a empirical evaluation, a potentially costly process that, however, has to be performed only once for a given scenario.

6. Case study: Service discovery and access

After the in-depth evaluation presented in the previous section, we now describe the application of the techniques proposed in this paper to a more realistic and compelling case study: *service discovery and access in a peer-to-peer network*. In particular, the main goal of the case study is to demonstrate that the proposed techniques can support a non-trivial scenario where:

- devices initiate/terminate/participate to CCPs depending on their current status determined by a finite state machine (FSM);
- each device can participate to several processes at the same time;
- processes have a complex lifetime: starting their life with a spherical wave expansion, they continue with several simultaneous tree-based expansions springing from different sources, and finally they end their life due to a corresponding number of tree-based terminations.

The source code, build infrastructure, and instructions for running the case study are also publicly available at the provided permanent repository (cf. Section 5 and [33]).

6.1. Scenario

In this scenario, there is a network of nodes which provide *services* (e.g., computing, storage, or sensing services) and that can act as service *consumers* and/or service *providers*. For simplicity, we assume that each node in the network offers exactly one service in a predefined set $\{S_1, \ldots, S_K\}$. Moreover, each node associates a *rank* $r \in [0, 1)$ to its offered service S, with a higher rank corresponding to a better service.

Dynamically, a potential consumer device can launch a service discovery process responsible for gathering at the device the offers provided by the devices in its surrounding. Once the device has acquired the information, it can choose the desired service offer for actual access of the corresponding service.

Most specifically, the behaviour of each node is modelled as per the state machine in Fig. 5:

- Nodes start in *IDLE* state. Triggered by some condition (see below), during a round of execution a node n can send a *disco*very message to search for a service S. Node n then transitions to the *DISCO* state.
- Each node m in *IDLE* state that offers service S replies with an *offer* message to m, which includes the rank r of S, and transitions to the *OFFER* state.

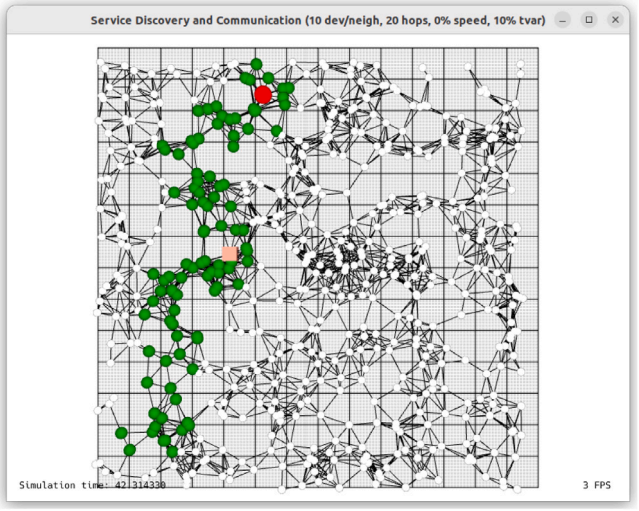

Fig. 6. Case study: a simple scenario with a square pink node sending an *ack* to the red node.

- Node n collects the offers and, after a *stabilisation* time elapses, the best-rank *offer* is accepted by sending an *ack* message to its sender m, while n transitions to the *SERVED* state. The other offers, if any, are simply ignored. Moreover, if node n does not receive an offer for service S within a specified *timeout* interval after sending its *disco* message, it returns to the *IDLE* state.
- Upon receiving the *ack*nowledgement for its offer, node m transitions to the *SERVING* state and starts sending a file to n, namely a sequence of N *data* messages at the rate of one message per round. If node m does not receive an ack within a specified interval after sending its *offer* message, it times out and it returns to the *IDLE* state.
- After sending (resp. receiving) the last *EOF* message, m (resp. n) returns to the *IDLE* state

The *disco* messages are sent using the *spherical* topology, which broadcasts them to all the nodes in the network. Moreover, we adopt the *wispp* termination strategy, since the process that delivers the message should be active in each node just for the time needed to propagate it to nodes further away. The other types of messages (*offer*, *ack*, and *data*) are instead sent using the *tree* topology, since they have a specific destination and thus benefit from a focused propagation along a spanning tree. We adopt the *ispp* termination strategy, which is particularly effective for the tree topology (Section 5.3.2). In order to limit the number of created processes, given a request by a node n, all the offers made for such a request share the same process, with pid equal to n. The same process is also used by n to send its *ack*nowledgement to the chosen server. Since in our case study a node n does not make other requests until the current one is served (or cancelled by a timeout), there is no ambiguity about which request is associated with process pid n.

Fig. 6 shows a snapshot of the FCPP graphical user interface during the execution of an instance of the case study. The white and coloured circles and squares represent the network devices, and neighbour nodes are connected with solid lines. In this instance, which is particularly simple, one node n (pink square) has sent a *disco* message for a service that was offered only by the red node m near the top. Node m has already sent an *offer* to m (its red colour denotes it is in *OFFER* status), which has been received by n; and node n has just sent an *ack* message towards m (its pink colour means it is in *SERVED* status). It is worth noting also the green nodes, that represent the intermediate nodes connecting n and m in the spanning tree. They are coloured because they host the active process spawned by m to send its offer to n, and

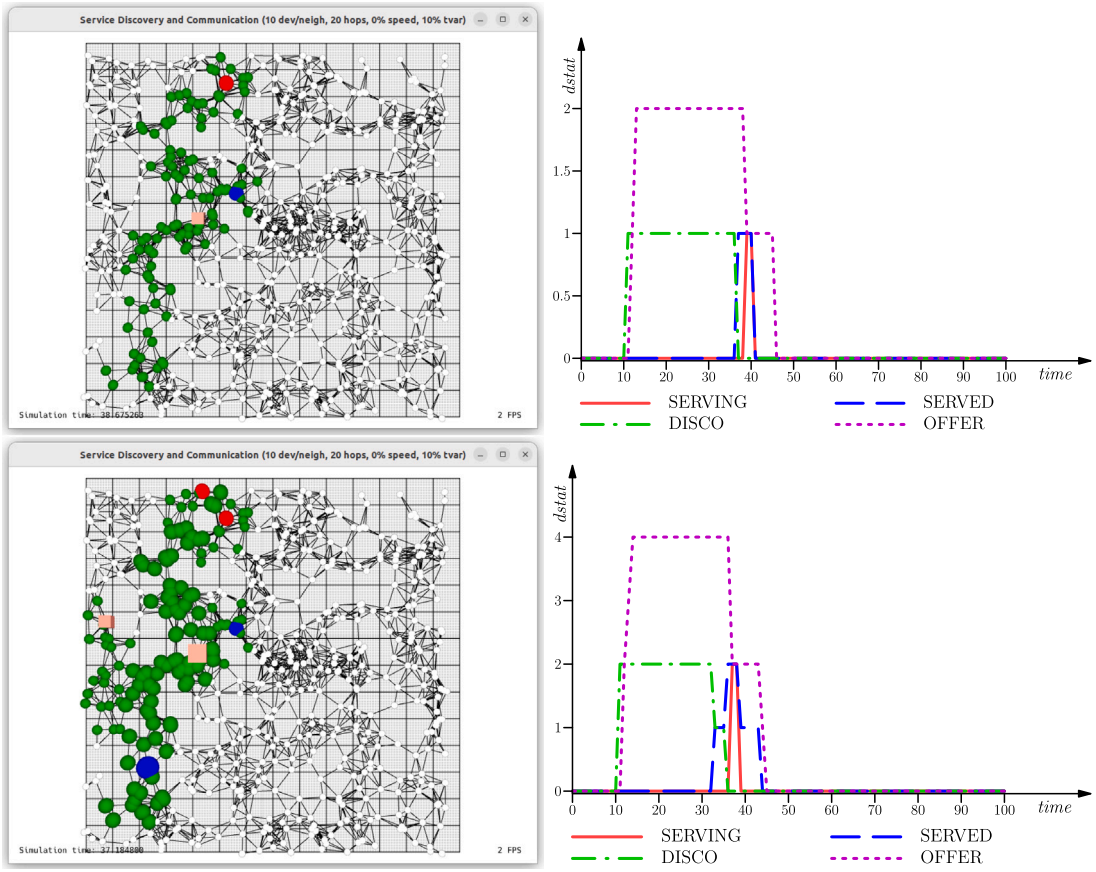

Fig. 7. Case study: other scenarios.

currently exploited by n to send its ack to m. Their green colour denotes the fact that they are in *IDLE* status, since they act just as the medium for the exchange of messages between n and m. In Section 6.3 we shall describe two slightly more complex scenarios of the case study execution.

6.2. Parameters

The case study is configured with the parameters discussed in Section 5.2.2 (i.e., temporal variance *tvar*, network density *dens*, network diameter *hops*, and device *speed*), plus the following:

1. *svc-types*, representing the number of different types of services offered by the nodes in the network. Each node offers exactly one of such services, identified by an *id* between 0 and (*svc-types* − 1).
2. *to-coeff*, the coefficient to be multiplied by *hops* for determining the timeout value used by the behaviour state-machine. For instance, if *hops* is 20 and *to-coeff* is 2, a node that has sent a *disco* message gives up if it does not receive an *offer* within $2\times20 = 40$ rounds.
3. *stab-coeff*, the coefficient to be multiplied by *hops* for determining the stabilisation time value for collecting offers used by the behaviour state-machine.

Unless otherwise noted, in the following section we have adopted the following configuration: $interval = 1$, $radius = 100$, $tvar = 10$, $dens = 10$, $hops = 20$, $speed = 0$, *svc-types* $= 100$, *to-coeff* $= 2$, and *stab-coeff* $= 1$.

6.3. More scenarios

Fig. 7 (top left) shows a screenshot of a case study execution similar to the one of Fig. 6, with the difference that the requested service is offered by two nodes: the red one as in the previous example, and the blue one. Since the ranks associated with the service by the two nodes are, respectively, 0.08 and 0.27, the second one is chosen by the requester. At the time of the screenshot, the requester is being *SERVED* (pink) and the chosen server is *SERVING* it (blue). The sequence of state transitions is illustrated in Fig. 7 (top right), where the total number of nodes in each state (except for *IDLE*) is plotted as a function of time. It is interesting to note that, shortly after the request is created at time 10, the two server nodes switch to the *OFFER* state almost at the same time. The chosen server goes to *SERVING* and then *IDLE* around time 40, while the other server returns to *IDLE* around time 46 (because it times out).

A last example is shown in Fig. 7 (bottom left). Here, two requesters (pink) are being *SERVED* by their respective best-rank servers (blue). The other servers are still in *OFFER*ing state, waiting for the timeout. What is particularly interesting to observe in this plot is the size of the nodes: in fact, some are clearly larger than others. In our FCPP simulation of the case study, we use the node size as a way to visualise how many processes are active in a node. In this relatively complex scenario, where several requesters are connected with several servers, some nodes host three active processes at the time of the screenshot. The sequence of state transitions is illustrated in Fig. 7 (bottom right).

7. Related work

In the following, we distinguish and relate upon multiple clusters of works that share some commonalities with our approach to distributed computational processes. The works that are most related, for both

methods and goals, are those on field-based coordination and ensemble-based approaches. However, we also mention pattern formation, swarm clustering, and epidemic network processes as sources of related ideas, to be further investigated in future work.

Field-based coordination, aggregate computing, and aggregate processes

This work takes inspiration from the framework of *aggregate computing* [13], where distributed systems are programmed "as wholes" through a network-wide program manipulating *(computational) fields* [13,60]. Specifically, we consider the notion of an *aggregate process* [15,31], which models a transient, concurrent aggregate computation running on a dynamic set of devices. Aggregate processes have been adopted in scenarios like swarm-based exploration [15], peer-to-peer messaging [15], service discovery [15], multi-agent plan repairing [61], and spatial coordination [21]. Aggregate processes have been formalised both in the field calculus [13,15,31] and in the exchange calculus [62–65]. In this work, we provide a more general characterisation of aggregate processes, that does not require to understand the semantics details of field and exchange calculi. Moreover, we extend the state of the art on aggregate processes [15] with more efficient and sophisticated policies for controlling their lifecycle.

A related tuple-based coordination approach is *Tuples On The Air (TOTA)* [66]. In TOTA, tuples propagate and evolve (e.g., their content) in a network, according to various kinds of application-specific rules: *propagation rules*, controlling where tuples may reside and how they transform; and *maintenance rules*, controlling how tuples change over time or w.r.t. environmental events. Our CCPs are a more general abstraction, and can be used to implement TOTA tuples as well as TOTA middleware activities—following the approach in [21] where aggregate processes are used to support a decentralised coordination model with situated tuples.

Pattern languages

The proposed work can be framed within the field of self-organising multi-robot pattern formation [67]. The survey [11] on spatial computing identifies *pattern languages* as a class of works aiming to produce spatial, geometrical, or topological patterns in amorphous computers made of several simple, unreliable devices locally communicating with one another. For instance, Origami Shape Language (OSL) [68], enables to build shapes on a surface through a sequence of flat folding operations. An OSL program is then implemented by uniform cell programs leveraging gradients, neighbourhood queries, and other local operations like local folding. Another example is Growing Point Language [69], which uses trajectories of "growing points" (mobile computations) diffusing across nodes to form patterns. However, unlike collective processes, growing points are active at a single domain at a time. These works are related as they propose mechanisms for building shapes incrementally in systems of neighbour-interacting devices. However, they tend to focus on the shape of groups and neglect the computation carried out in the defined domains. However, generally speaking, as shown in recent reviews on multi-robot pattern formation [67], much of the emphasis of the research area is on microscopic models and algorithms rather than on macro-programming solutions. One recent field-based domain-specific library supporting *macro-level programming* of pattern formation is *Macro-Swarm* [70]: indeed, it internally leverages aggregate processes (i.e. CCPs) for the dynamics of several building blocks, hence emphasising the significance of the techniques discussed in this paper.

Ensemble-based approaches

An *ensemble* is a dynamic group of devices that forms to support group-level tasks. In *Distributed Emergent Ensembles of Components (DEECo)* [14], ensembles are characterised by a membership condition that expresses how a set of components get bound together. Within an ensemble, the components interact by implicit knowledge exchange. Our collective processes can also be seen as regulated through a membership condition, i.e., the status determining whether the node is willing to participate in the process; however, ensemble formation is a dynamic activity that runs on a given communication topology that also regulates interactions within processes. *Service Component Ensemble Language (SCEL)* [26] is a language that enables to express the behaviour of ensembles interacting via attribute-based communication. Ensemble formation is thus regulated through predicates over attributes exposed by components. This is different from our collective processes, where communication is constrained by both the given neighbouring relationship and process membership. In summary, both in DEECo and SCEL, the key aspect of ensemble domain propagation and shrinking addressed in this paper is not directly captured.

Clustering and area formation

Swarm clustering [71] brings the data clustering problem into swarm settings. The idea is to group agents into *clusters* such that the agents in the same cluster are more correlated to each other (e.g., spatially or temporally) than to the agents belonging to other clusters. For instance, in [72], a mathematical model for cluster-based group formation is proposed that takes inspiration from bee foraging and recruitment in order to assemble groups with complementary skills. A similar problem involves organising a system into regions or areas to solve a certain problem with a configurable level of decentralisation [23], cf. the *Self-organising Coordination Regions* pattern [24]. Swarm clusters and such pattern instances can, indeed, be expressed as collective processes. Also, collective processes can seamlessly model the case where clusters need to overlap, which may be instrumental for conflict resolution or inter-regional coordination. Vice versa, clustering processes could be used to regulate the formation of collective process domains; however, these could not naturally cover all the possible evolution dynamics that CASs may exhibit (e.g., wave-like ones).

Spreading and epidemic processes over networks

The topic of this paper is also potentially related to spreading and epidemic processes in time-varying and complex networks [73,74]. Among the key distinguishing factors between those works and this one there are the system model (cf. Section 3) and the emphasis on programmability of the logic for incremental process domain evolution. However, studying the dynamics of collective processes via tools and methods from network science could be an interesting future work.

8. Conclusion

8.1. Discussion

In this manuscript, we investigate algorithmic techniques for the dynamic evolution of collective computational processes and ensembles, leveraging descriptions over event structures and simulations. Starting from a general formalisation of the field-based framework of aggregate processes [15,21,31], we propose algorithms for controlling the effective propagation and termination of group-wise processes, enabling trade-offs in terms of efficiency (cf., messages exchanges and rounds of execution), functionality, and design complexity. Specifically, we leverage statistics of information speed to devise novel strategies for wave-like propagation of processes and their shrinking (up to completion). Then, we show by simulation that the proposed algorithms improve over the state of the art [15,31,64,65] and run a sensitivity analysis to study their behaviour in different network settings.

In summary, we have shown how a simple mechanism to express the participation of a device in a process (cf. status values *internal*, *external*, and *border*), when supported by collaborative algorithms, enables to capture various patterns of ensemble evolution (including growth to seek more devices and termination to close a collaboration). Ultimately, we find that CCPs offer a fine-grained way to control the formation of

ensembles and that our formal framework provides a setting where the evolution of ensembles and the relationships between shape and computation can be investigated (cf. formal properties in Section 4.5). We remark that the approach is most applicable in large-scale ecosystems where multiple collective tasks are to be carried out by dynamic teams of devices: similar ideas have been applied in application scenarios like environment exploration by swarms [15], flood monitoring for emergency management [42], and smart warehouse management [52].

8.2. Future work

In the future, we consider analysing CCPs with methods found in fields like complex networks and epidemics, e.g., to devise formal properties about the dynamics of process evolution. Furthermore, we plan to apply the algorithms proposed in this work to various coordination patterns and scenarios, such as the *Self-organising Coordination Regions* pattern [24] and models based on *situated tuples* [21]. In addition, more advanced CCPs abstractions could be investigated, for instance, of tasks that follow specific state machines. Finally, a still open challenge is whether it is possible to realise a process propagation strategy that combines the efficiency of the tree topology with the robustness of the spherical topology. Recent insights on distributed data collection [49] might help towards this purpose, but more research is needed before any conclusion is drawn. In general, more process propagation patterns could be developed, tuned for specific purposes and providing various trade-offs, and possibly exploiting differentiated messages to neighbours as discussed in [62–65].

CRediT authorship contribution statement

Giorgio Audrito: Writing – review & editing, Writing – original draft, Validation, Software, Investigation, Formal analysis, Conceptualization. **Roberto Casadei:** Writing – review & editing, Writing – original draft, Project administration, Methodology, Conceptualization. **Gianluca Torta:** Writing – review & editing, Writing – original draft, Validation, Software, Investigation, Conceptualization.

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.

Data availability

We have published data/code on Zenodo: https://doi.org/10.5281/zenodo.8310743.

Acknowledgements

This work has been partially supported by the Italian PRIN project "COMMON-WEARS" (2020HCWWLP) and the EU/MUR FSE REACT-EU PON R&I 2014–2020. The work is also part of the project NODES which has received funding from the MUR – M4C2 1.5 of PNRR with grant agreement no. ECS00000036.

References

[1] S.S. Goel, A. Goel, M. Kumar, G. Moltó, A review of Internet of Things: qualifying technologies and boundless horizon, J. Reliab. Intell. Environ. 7 (1) (2021) 23–33, http://dx.doi.org/10.1007/S40860-020-00127-W.

[2] R.D. Nicola, S. Jähnichen, M. Wirsing, Rigorous engineering of collective adaptive systems: special section, Int. J. Softw. Tools Technol. Transf. 22 (4) (2020) 389–397, http://dx.doi.org/10.1007/s10009-020-00565-0.

[3] A. Bucchiarone, M. D'Angelo, D. Pianini, G. Cabri, M. De Sanctis, M. Viroli, R. Casadei, S. Dobson, On the social implications of collective adaptive systems, IEEE Technol. Soc. Mag. 39 (3) (2020) 36–46, http://dx.doi.org/10.1109/MTS.2020.3012324.

[4] O. Inverso, C. Trubiani, E. Tuosto, Abstractions for collective adaptive systems, in: Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Proceedings, Part II, in: LNCS, Vol. 12477, Springer, 2020, pp. 243–260, http://dx.doi.org/10.1007/978-3-030-61470-6_15.

[5] R. Casadei, Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling, ACM Comput. Surv. 55 (13s) (2023) http://dx.doi.org/10.1145/3579353.

[6] T.W. Malone, K. Crowston, The interdisciplinary study of coordination, ACM Comput. Surv. 26 (1) (1994) 87–119, http://dx.doi.org/10.1145/174666.174668.

[7] J. Ferber, Multi-agent Systems - An Introduction to Distributed Artificial Intelligence, Addison-Wesley-Longman, 1999.

[8] S. von Mammen, S. Tomforde, J. Hähner, An organic computing approach to self-organizing robot ensembles, Front. Robot. AI 3 (2016) 67, http://dx.doi.org/10.3389/frobt.2016.00067.

[9] M. Brambilla, E. Ferrante, M. Birattari, M. Dorigo, Swarm robotics: a review from the swarm engineering perspective, Swarm Intell. 7 (1) (2013) 1–41, http://dx.doi.org/10.1007/s11721-012-0075-2.

[10] R. Newton, M. Welsh, Region streams: Functional macroprogramming for sensor networks, in: Workshop on Data Management for Sensor Networks, 2004, pp. 78–87, http://dx.doi.org/10.1145/1052199.1052213.

[11] J. Beal, S. Dulman, K. Usbeck, M. Viroli, N. Correll, Organizing the aggregate: Languages for spatial computing, in: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, IGI Global, 2013, pp. 436–501, http://dx.doi.org/10.4018/978-1-4666-2092-6.ch016.

[12] T.D. Wolf, T. Holvoet, Designing self-organising emergent systems based on information flows and feedback-loops, in: Proceedings of the 1st International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, IEEE Computer Society, 2007, pp. 295–298, http://dx.doi.org/10.1109/SASO.2007.16.

[13] 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) http://dx.doi.org/10.1016/j.jlamp.2019.100486.

[14] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, DEECO: an ensemble-based component system, in: CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering, ACM, 2013, pp. 81–90, http://dx.doi.org/10.1145/2465449.2465462.

[15] R. Casadei, M. Viroli, G. Audrito, D. Pianini, F. Damiani, Engineering collective intelligence at the edge with aggregate processes, Eng. Appl. Artif. Intell. 97 (2021) 104081, http://dx.doi.org/10.1016/j.engappai.2020.104081.

[16] Z. Wood, A. Galton, A taxonomy of collective phenomena, Appl. Ontol. 4 (3–4) (2009) 267–292, http://dx.doi.org/10.3233/AO-2009-0071.

[17] O. Scekic, T. Schiavinotto, S. Videnov, M. Rovatsos, H.L. Truong, D. Miorandi, S. Dustdar, A programming model for hybrid collaborative adaptive systems, IEEE Trans. Emerg. Top. Comput. 8 (1) (2020) 6–19, http://dx.doi.org/10.1109/TETC.2017.2702578.

[18] X. Chen, P. Zhang, G. Du, F. Li, A distributed method for dynamic multi-robot task allocation problems with critical time constraints, Robot. Auton. Syst. 118 (2019) 31–46, http://dx.doi.org/10.1016/j.robot.2019.04.012.

[19] I. Navarro, F. Matía, A survey of collective movement of mobile robots, Int. J. Adv. Robot. Syst. 10 (1) (2013) 73, http://dx.doi.org/10.5772/54600.

[20] S. Mariani, A. Omicini, Space-aware coordination in respect, in: M. Baldoni, C. Baroglio, F. Bergenti, A. Garro (Eds.), 14th Workshop "from Objects To Agents", Proceedings, in: CEUR Workshop Proceedings, Vol. 1099, CEUR-WS.org, 2013, pp. 1–7.

[21] R. Casadei, M. Viroli, A. Ricci, G. Audrito, Tuple-based coordination in large-scale situated systems, in: Coordination Models and Languages - 23rd International Conference, COORDINATION 2021, Proceedings, in: LNCS, Vol. 12717, Springer, 2021, pp. 149–167, http://dx.doi.org/10.1007/978-3-030-78142-2_10.

[22] R. Casadei, S. Mariani, D. Pianini, M. Viroli, F. Zambonelli, Space-fluid adaptive sampling: A field-based, self-organising approach, in: M.H. ter Beek, M. Sirjani (Eds.), Coordination Models and Languages - 24th International Conference, COORDINATION 2022, Proceedings, in: LNCS, Vol. 13271, Springer, 2022, pp. 99–117, http://dx.doi.org/10.1007/978-3-031-08143-9_7.

[23] D. Weyns, T. Holvoet, Regional synchronization for simultaneous actions in situated multi-agent systems, in: Multi-Agent Systems and Applications III, CEEMAS 2003, Proceedings, in: LNCS, Vol. 2691, Springer, 2003, pp. 497–510, http://dx.doi.org/10.1007/3-540-45023-8_48.

[24] R. Casadei, D. Pianini, M. Viroli, A. Natali, Self-organising coordination regions: A pattern for edge computing, in: COORDINATION'19, Proceedings, in: LNCS, Vol. 11533, Springer, 2019, pp. 182–199, http://dx.doi.org/10.1007/978-3-030-22397-7_11.

[25] C. Pinciroli, G. Beltrame, Buzz: A programming language for robot swarms, IEEE Softw. 33 (4) (2016) 97–100, http://dx.doi.org/10.1109/MS.2016.95.

[26] R.D. Nicola, M. Loreti, R. Pugliese, F. Tiezzi, A formal approach to autonomic systems programming: The SCEL language, ACM Trans. Auton. Adapt. Syst. 9 (2) (2014) 7:1–7:29, http://dx.doi.org/10.1145/2619998.

[27] G. Audrito, R. Casadei, G. Torta, On the dynamic evolution of distributed computational aggregates, in: IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion, ACSOS-C 2022, IEEE, 2022, pp. 37–42, http://dx.doi.org/10.1109/ACSOSC56246.2022.00024.

[28] R. Casadei, E.D. Nitto, I. Gerostathopoulos, D. Pianini, I. Dusparic, et al. (Eds.), IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion, ACSOS-C 2022, Virtual, CA, USA, September 19-23, 2022, IEEE, 2022, http://dx.doi.org/10.1109/ACSOS-C56246.2022.

[29] M. Nielsen, G.D. Plotkin, G. Winskel, Petri nets, event structures and domains, part I, Theoret. Comput. Sci. 13 (1981) 85–108, http://dx.doi.org/10.1016/0304-3975(81)90112-2.

[30] G. Audrito, J. Beal, F. Damiani, M. Viroli, Space-time universality of field calculus, in: COORDINATION'18, in: LNCS, Vol. 10852, Springer, 2018, pp. 1–20, http://dx.doi.org/10.1007/978-3-319-92408-3_1.

[31] R. Casadei, M. Viroli, G. Audrito, D. Pianini, F. Damiani, Aggregate processes in field calculus, in: COORDINATION'19, Proceedings, Springer, 2019, pp. 200–217, http://dx.doi.org/10.1007/978-3-030-22397-7_12.

[32] 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.

[33] G. Torta, G. Audrito, R. Casadei, fcpp-experiments/process-management: 1.0, 2023, http://dx.doi.org/10.5281/zenodo.8310743.

[34] G.D. Abowd, Beyond weiser: From ubiquitous to collective computing, Computer 49 (1) (2016) 17–23, http://dx.doi.org/10.1109/MC.2016.22.

[35] J. Hendler, T. Berners-Lee, From the semantic web to social machines: A research challenge for AI on the World Wide Web, Artificial Intelligence 174 (2) (2010) 156–161, http://dx.doi.org/10.1016/j.artint.2009.11.010.

[36] M.A. Jamshed, K. Ali, Q.H. Abbasi, M.A. Imran, M. Ur-Rehman, Challenges, applications, and future of wireless sensors in internet of things: A review, IEEE Sens. J. 22 (6) (2022) 5482–5494, http://dx.doi.org/10.1109/jsen.2022.3148128.

[37] S. Iftikhar, S.S. Gill, C. Song, M. Xu, M.S. Aslanpour, A.N. Toosi, J. Du, H. Wu, S. Ghosh, D. Chowdhury, M. Golec, M. Kumar, A.M. Abdelmoniem, F. Cuadrado, B. Varghese, O.F. Rana, S. Dustdar, S. Uhlig, AI-based fog and edge computing: A systematic review, taxonomy and future directions, Int. Things 21 (2023) 100674, http://dx.doi.org/10.1016/J.IOT.2022.100674.

[38] R. Casadei, Artificial collective intelligence engineering: A survey of concepts and perspectives, Artif. Life (2023) 1–35, http://dx.doi.org/10.1162/artl_a_00408.

[39] T.M. Mengistu, D. Che, Survey and taxonomy of volunteer computing, ACM Comput. Surv. 52 (3) (2019) 59:1–59:35, http://dx.doi.org/10.1145/3320073.

[40] S. Berman, Á.M. Halász, M.A. Hsieh, V. Kumar, Optimized stochastic policies for task allocation in swarms of robots, IEEE Trans. Robot. 25 (4) (2009) 927–937, http://dx.doi.org/10.1109/TRO.2009.2024997.

[41] G. Aguzzi, G. Audrito, R. Casadei, F. Damiani, G. Torta, M. Viroli, A field-based computing approach to sensing-driven clustering in robot swarms, Swarm Intell. (2022) http://dx.doi.org/10.1007/s11721-022-00215-y.

[42] G. Aguzzi, R. Casadei, D. Pianini, M. Viroli, Dynamic decentralization domains for the internet of things, IEEE Int. Comput. (2022) 1–10, http://dx.doi.org/10.1109/mic.2022.3216753.

[43] R. Casadei, D. Pianini, A. Placuzzi, M. Viroli, D. Weyns, Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment, Future Int. 12 (11) (2020) 203, http://dx.doi.org/10.3390/fi12110203.

[44] S. Garnier, N. Ross, B. Rudis, M. Sciaini, C. Scherer, Viridis: Default color maps from 'matplotlib', 1, 2018, R package v0.5.

[45] J. Xu, W. Liu, F. Lang, Y. Zhang, C. Wang, Distance measurement model based on RSSI in WSN, Wirel. Sens. Netw. 2 (8) (2010) 606–611, http://dx.doi.org/10.4236/wsn.2010.28072.

[46] K. Benkic, M. Malajner, P. Planinsic, Z. Cucej, Using RSSI value for distance estimation in wireless sensor networks based on ZigBee, in: 15th International Conference on Systems, Signals and Image Processing, 2008, pp. 303–306, http://dx.doi.org/10.1109/IWSSIP.2008.4604427.

[47] J.L. Fernandez-Marquez, G.D. 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.

[48] Y. Mo, S. Dasgupta, J. Beal, Robustness of the adaptive Bellman-Ford algorithm: Global stability and ultimate bounds, IEEE Trans. Autom. Control. 64 (10) (2019) 4121–4136, http://dx.doi.org/10.1109/TAC.2019.2904239.

[49] G. Audrito, R. Casadei, F. Damiani, D. Pianini, M. Viroli, Optimal resilient distributed data collection in mobile edge environments, Comput. Electr. Eng. 96 (Part) (2021) 107580, http://dx.doi.org/10.1016/j.compeleceng.2021.107580.

[50] R. Casadei, M. Viroli, G. Aguzzi, D. Pianini, ScaFi: A scala DSL and toolkit for aggregate programming, SoftwareX 20 (2022) 101248, http://dx.doi.org/10.1016/j.softx.2022.101248.

[51] D. Pianini, R. Casadei, M. Viroli, Self-stabilising priority-based multi-leader election and network partitioning, in: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2022, IEEE, 2022, pp. 81–90, http://dx.doi.org/10.1109/ACSOS55765.2022.00026.

[52] L. Testa, G. Audrito, F. Damiani, G. Torta, Aggregate processes as distributed adaptive services for the Industrial Internet of Things, Pervasive Mob. Comput. 85 (2022) 101658, http://dx.doi.org/10.1016/j.pmcj.2022.101658.

[53] G. Audrito, J. Beal, F. Damiani, D. Pianini, M. Viroli, Field-based coordination with the share operator, Log. Methods Comput. Sci. 16 (4) (2020) http://dx.doi.org/10.23638/LMCS-16(4:1)2020.

[54] G. Audrito, FCPP: an efficient and extensible field calculus framework, in: International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), IEEE, 2020, pp. 153–159, http://dx.doi.org/10.1109/ACSOS49614.2020.00037.

[55] J. Beal, Flexible self-healing gradients, in: ACM Symposium on Applied Computing, Proceedings, 2009, pp. 1197–1201.

[56] 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.

[57] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426, http://dx.doi.org/10.1145/362686.362692.

[58] K. Park, S. Pack, T. Kwon, Proximity based peer-to-peer overlay networks (P3ON) with load distribution, in: Information Networking. Towards Ubiquitous Networking and Services: International Conference, ICOIN 2007, Estoril, Portugal, January 23-25, 2007. Revised Selected Papers, Springer, 2008, pp. 234–243.

[59] K. Haseeb, I. Ud Din, A. Almogren, N. Islam, A. Altameem, RTS: A robust and trusted scheme for IoT-based mobile wireless mesh networks, IEEE Access 8 (2020) 68379–68390, http://dx.doi.org/10.1109/ACCESS.2020.2985851.

[60] M. Mamei, F. Zambonelli, L. Leonardi, Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems, in: 3rd International Workshop on Engineering Societies in the Agents World, in: LNCS, Vol. 2577, Springer, 2002, pp. 68–81, http://dx.doi.org/10.1007/3-540-39173-8_6.

[61] G. Audrito, R. Casadei, G. Torta, Fostering resilient execution of multi-agent plans through self-organisation, in: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2021, Companion Volume, IEEE, 2021, pp. 81–86, http://dx.doi.org/10.1109/ACSOS-C52956.2021.00076.

[62] G. Audrito, R. Casadei, F. Damiani, G. Salvaneschi, M. Viroli, The exchange calculus (XC): a functional programming language design for distributed collective systems, J. Syst. Softw. 210 (2024) 111976, http://dx.doi.org/10.1016/J.JSS.2024.111976.

[63] G. Audrito, R. Casadei, F. Damiani, G. Salvaneschi, M. Viroli, Functional programming for distributed systems with XC, in: 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany, in: LIPIcs, Vol. 222, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 20:1–20:28, http://dx.doi.org/10.4230/LIPICS.ECOOP.2022.20.

[64] G. Audrito, R. Casadei, F. Damiani, G. Torta, M. Viroli, Programming distributed collective processes for dynamic ensembles and collective tasks, in: COORDINATION'23, Proceedings, in: LNCS, Vol. 13908, Springer, 2023, pp. 71–89, http://dx.doi.org/10.1007/978-3-031-35361-1_4.

[65] G. Audrito, R. Casadei, F. Damiani, G. Torta, M. Viroli, Programming distributed collective processes in the exchange calculus, 2024, http://dx.doi.org/10.48550/ARXIV.2401.11212, CoRR arXiv:2401.11212.

[66] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications: The TOTA approach, ACM Trans. Softw. Eng. Methodol. 18 (4) (2009) 15:1–15:56, http://dx.doi.org/10.1145/1538942.1538945.

[67] H. Oh, A.R. Shirazi, C. Sun, Y. Jin, Bio-inspired self-organising multi-robot pattern formation: A review, Robot. Auton. Syst. 91 (2017) 83–100, http://dx.doi.org/10.1016/j.robot.2016.12.006.

[68] R. Nagpal, Programmable Self-Assembly: Constructing Global Shape Using Biologically-Inspired Local Interactions and Origami Mathematics (Ph.D. thesis), MIT (USA), 2001, URL http://hdl.handle.net/1721.1/86667.

[69] D. Coore, Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer (Ph.D. thesis), MIT (USA), 1999, URL http://hdl.handle.net/1721.1/80483.

[70] G. Aguzzi, R. Casadei, M. Viroli, MacroSwarm: A field-based compositional framework for swarm programming, in: Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held As Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings, in: Lecture Notes in Computer Science, Vol. 13908, Springer, 2023, pp. 31–51, http://dx.doi.org/10.1007/978-3-031-35361-1_2.

[71] C. Lee, M. Kim, S. Kazadi, Robot clustering, in: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, 2005, IEEE, 2005, pp. 1449–1454, http://dx.doi.org/10.1109/ICSMC.2005.1571350.

[72] D.S. dos Santos, A.L.C. Bazzan, Distributed clustering for group formation and task allocation in multiagent systems: A swarm intelligence approach, Appl. Soft Comput. 12 (8) (2012) 2123–2131, http://dx.doi.org/10.1016/j.asoc.2012.03.016.

[73] C. Nowzari, V.M. Preciado, G.J. Pappas, Analysis and control of epidemics: A survey of spreading processes on complex networks, IEEE Control Syst. Mag. 36 (1) (2016) 26–46, http://dx.doi.org/10.1109/MCS.2015.2495000.

[74] M. Cremonini, S. Maghool, The dynamical formation of ephemeral groups on networks and their effects on epidemics spreading, Sci. Rep. 12 (1) (2022) 1–10.

Giorgio Audrito is an assistant professor at Università degli Studi di Torino, Torino, Italy. His research interests include distributed computing, programming languages, distributed algorithms and graph algorithms; as well as innovative didactic methods through gamification. Since 2013 he is the team leader of the Italian team at the International Olympiad in Informatics. Since 2020 he is the original designer, main developer and maintainer of FCPP (http://fcpp.github.io), a C++ framework for aggregate programming, which is winner of a best artefact and an outstanding artefact awards.

Roberto Casadei is an assistant professor at Alma Mater Studiorum-Università di Bologna, Cesena, Italy. He has a Ph.D. in Computer Science & Engineering from the same university, with a thesis awarded by the IEEE TCSC. His research interests revolve around software engineering and distributed artificial intelligence. He has 50+ publications in international journals and conferences on topics including collective intelligence, aggregate computing, self-* systems, and IoT/CPS. He also leads the development of the open-source ScaFi aggregate programming toolkit. He has been serving in the organising and program committees of multiple conferences such as ACSOS, COORDINATION, ICCCI, and SAC, as a guest editor and reviewer for renowned international journals, and as editorial board member of JAISCR and Elsevier IoT.

Gianluca Torta is an assistant professor at Università degli Studi di Torino, Torino, Italy. His research interests include distributed computing and algorithms, embedded systems, and model-based reasoning (diagnosis and planning). Since 2021, he is a contributor of FCPP, a C++ framework for aggregate programming.