Internet of Things 28 (2024) 101412

Contents lists available at ScienceDirect

Internet of Things

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

Dynamic IoT deployment reconfiguration: A global-level self-organisation approach

Nicolas Farabegoli *, Danilo Pianini, Roberto Casadei, Mirko Viroli

Alma Mater Studiorum – University of Bologna, Cesena, Italy

ARTICLE INFO

Keywords:
Distributed systems
Edge–cloud continuum
Dynamic reconfiguration
Collective adaptive systems
Aggregate computing
Pulverisation
Distributed middleware

ABSTRACT

The edge–cloud continuum provides a heterogeneous, multi-scale, and dynamic infrastructure supporting complex deployment profiles and trade-offs for application scenarios like those found in the Internet of Things and large-scale cyber–physical systems domains. To exploit the continuum, applications should be designed in a way that promotes flexibility and reconfigurability, and proper management (sub-)systems should take care of reconfiguring them in response to changes in the environment or non-functional requirements. Approaches may leverage optimisation-based or heuristic-based policies, and decision making may be centralised or distributed: this work investigates decentralised heuristic-based approaches. In particular, we focus on the pulverisation approach, whereby a distributed software system is automatically partitioned ("pulverised") into different deployment units. In this context, we address two main research problems: how to support the runtime reconfiguration of pulverised systems, and how to specify decentralised reconfiguring policies by a global perspective. To address the first problem, we design and implement a middleware for pulverised systems separating infrastructural and application concerns. To address the second problem, we leverage aggregate computing and exploit self-organisation patterns to devise self-stabilising reconfiguration strategies. By simulating deployments on different kinds of complex infrastructures, we assess the flexibility of the pulverisation middleware design as well as the effectiveness and resilience of the aggregate computing-based reconfiguration policies.

1. Introduction

The term *Edge-to-Cloud Continuum (ECC)* [1] refers to the heterogeneous, multi-layer, integrated infrastructure that combines resources at the edge, at the cloud, and along their data path: access points, routers, edge servers, cloud servers, etc. The ECC aims to combine the benefits and mitigate the issues of both sides, hence supporting requirements of complex deployments (e.g., low latency, privacy, and resource elasticity) and enabling complex and dynamic trade-offs of cost, performance, and sustainability. As such, it appears to be a key infrastructure for applications found in scenarios like the Internet of Things (IoT) and large-scale Cyber–Physical Systems (CPSs); in this work, we especially focus on *Collective Adaptive Systems (CASs)* [2,3], namely collections of devices solving collaborative tasks in dynamic environments (cf. swarms, crowd of wearables, computing ecosystems). However, the heterogeneity, distributed nature, and dynamicity of the ECC pose new challenges for the engineering and deployment of applications [4].

In order to fully and opportunistically exploit the potential of the ECC, applications should be designed in decoupled modules or components [5–8], promoting flexibility in application deployment and management. Then, managing systems should enable the

* Corresponding author.
E-mail addresses: nicolas.farabegoli@unibo.it (N. Farabegoli), danilo.pianini@unibo.it (D. Pianini), roby.casadei@unibo.it (R. Casadei), mirko.viroli@unibo.it (M. Viroli).

https://doi.org/10.1016/j.iot.2024.101412
Received 5 April 2024; Received in revised form 11 October 2024; Accepted 20 October 2024
Available online 28 October 2024

reconfiguration of applications at runtime [7], for accommodating changes in the environment (infrastructure and/or requirements). Even better, to streamline maintenance, applications should be able to reconfigure *themselves*, i.e., they should exhibit *self-* properties* [9,10] such as self-adaptation, self-organisation, and self-(re)configuration.

In the literature, several component models and techniques have been proposed for application partitioning and reconfiguration [5,7]. In the context of the IoT and large-scale CPSs, and most specifically for CASs [2], the *pulverisation approach* [11] to application partitioning and deployment have proven especially suitable [12]. Consider, for instance, an IoT application whose goal is coordinating a set of situated smartphones and wearable devices, e.g., implementing a real-time game in the real world. We call these devices *application-level hosts*. Since peer-to-peer communication is hardly viable with the most common current mainstream technology, the application can exploit the networked infrastructure composed of edge servers and the cloud, which are transparent from the point of view of the business logic, representing *purely infrastructural hosts*. In the pulverisation approach, the whole system is partitioned into a collective of devices (one per each application-level host), and each device is further split into multiple deployment units capturing aspects like (i) sensing, (ii) actuation, and (iii) behaviour. The result is called a *pulverised system*. When the functionality of the system is determined in terms of the activity and interactions carried out by those deployment units, different allocations of them over the networked hosts can be produced, each one resulting in potentially different non-functional profiles and trade-offs, yet all retaining the same functional logic, as far as the underlying network is not segmented.

In this paper, we address the problem of managing the deployment of pulverised systems in the ECC, with specific focus on the runtime reconfiguration of the system, and, particularly, on techniques to express runtime reconfiguration rules from a global stance, observing and controlling the mapping of the application over the ECC as if the latter were a single entity. To this end, we provide an extension to the pulverisation framework that enables the specification of deployment reconfiguration rules through the *Aggregate Computing* paradigm [13], exposing the whole ECC as a single manifold. To support such extension, we devise a novel architectural model capturing the separation between the physical (underlying, among devices) and logical (among application-level hosts) networks, of which we demonstrate the viability by implementing a practical middleware.

In a nutshell, the main contributions of this paper are:

1. a middleware architecture for managing pulverised systems in the ECC;
2. the use of *Aggregate Computing* as a possible solution to manage the dynamic reconfiguration of the system;
3. an evaluation of the proposed reconfiguration approach comparing the adaptive solution with a non-adaptive counterpart, with a corresponding open-source, archived artefact [14].

The rest of this paper is organised as follows. Section 2 clarifies the scope and provides research questions. Section 3 provides a brief overview of the Aggregate Computing paradigm and the pulverisation approach. Section 4 describes the proposed middleware architecture for engineering CASs in the ECC. Section 5 describes the proposed reconfiguration approach based on Aggregate Computing. Section 6 presents the evaluation of the approach in terms of QoS metrics. Section 7 discusses related work. Finally, Section 8 concludes the paper and outlines future work.

2. Research questions

The complexity of modern infrastructures like the ECC poses new challenges to the design and deployment of distributed systems and applications. The pulverisation approach has proven an effective solution to modular design of CASs [11], enabling deployers to come up with different deployment plans to be evaluated and enacted [12]. However, prior work has not investigated how pulverised systems should be managed at runtime, namely how a *middleware* for them should be organised and work, and how these could be reconfigured at runtime to improve their non-functional profile in response to changes in the application requirements or to unpredicted conditions of the underlying infrastructure. In particular, to address potentially *very large-scale* systems [2], we investigate decentralised and heuristic-based reconfiguration solutions (rather than centralised and optimisation-based solutions). Though automatic design approaches for reconfiguration policies could be adopted [15], e.g., based on learning (cf. multi-agent reinforcement learning [16]), in this work we focus on engineered specification-based approaches, and especially on the feasibility of using a *macro-programming* [17] approach like *aggregate computing* [13,18] to support the definition of reconfiguration policies by a *global* perspective.

In this work, we aim to answer the following research questions:

(RQ1) How can a pulverised system be effectively deployed and managed in complex infrastructures like the Edge-to-Cloud Continuum?
(RQ2) How can a pulverised system be dynamically reconfigured in a decentralised way using global policies?
(RQ3) What are the benefits and limitations of using global, decentralised policies?

To answer these questions, we propose a middleware architecture for engineering CASs in the ECC supporting the pulverisation approach, and the use of the Aggregate Computing paradigm as a possible solution to specify distributed and global reconfiguration policies.

3. Background

3.1. Aggregate computing

Aggregate computing [18] is a development approach for programming CASs. It comes with a formal basis, captured by *field calculi* [18], concrete programming language implementations (like ScaFi [19]), and tools (like the Alchemist simulator [20]). Aggregate computing has been used in the past to program self-adaptive systems in the context of the IoT and the ECC. Application scenarios include crowd detection and management [13], smart cities [11], contact tracing [21], and smart warehouses [22], among the others.

To provide a more concrete and detailed example, in the context of *smart traffic management*, aggregate computing can be used to manage the city traffic flow, where each car shares its GPS positioning, and the system regulates the traffic lights to limit congestions. During an emergency (e.g. ambulance deployment), the system can promptly react by managing the traffic light, and signalling the emergency to the cars suggesting appropriate movement. Aggregate computing can be adopted for managing the traffic and coordinate the cars avoiding traffic congestions, especially during an emergency event. In this scenario the ECC can be opportunistically exploited: during normal condition the cars host the computation of the aggregate program interacting in a peer-to-peer fashion; while the edge nodes collect traffic sensors and cameras data to manage the traffic lights. During an emergency, to ensure low-latency, the aggregate programs' execution is moved into the appropriate edge node(s) (physically located near the emergency region) ensuring faster communication, and send back to the cars only the final, computed instructions. The cloud can be exploited to collect data, and improve the traffic management algorithm with the new collected data from the edge nodes. Thanks the ECC the system can be dynamically reconfigured to ensure the best performance at time. Understanding aggregate computing boils down to understanding two main, synergic pieces: the system model that defines how execution unfolds, and the programming model that defines how the behavioural logic is expressed.

3.1.1. System model

We define an *aggregate system* as a collection of *(computational) devices* organised as follows. A device has *sensors* and *actuators*—their only interface to the local environment. A device can interact only with a dynamic subset of other devices, called its *neighbours*. Each device executes in *asynchronous sense–compute–act rounds*:

1. *sense*: the device acquires its local context by *sampling sensors* and getting the *most recent unexpired message* from all the neighbours;
2. *compute*: the device executes the program (see Section 3.1.2) against its local context;
3. *act*: the program's output defines what *actuations* are performed and what data is *shared* with neighbours.

This model abstracts from several details which may be configured on a per-application basis: neighbouring relationship, round frequency, message expiration, etc.

3.1.2. Programming model

The developer writes a single program for the entire system: this is called an *aggregate program* and is evaluated by all the devices in asynchronous rounds as discussed previously. Denotationally, the programming model leverages the abstraction of a *computational field* [18,23], namely a map from a domain of device to computational values. So, for example, the act of reading the temperature sensor, when performed by all the devices, yields a (dynamic) field of temperatures. The field calculus [18] and its implementations like ScaFi [19] are *functional languages*: so, denotationally, programs consist of functions accepting fields as inputs and producing fields as output (while, operationally, on a single device, the functions operate on "plain old" local values or fields restricted to the neighbourhood). Each function may embed both computation and communication (by "marking" values to be shared—as actual communications happen according to the execution model), hence a function call may be used to activate a collective behaviour. Examples will be provided in Section 5. Further details, not fundamental for understanding the paper, can be found in referenced work [18].

3.1.3. Aggregate computing for reconfiguration: motivation

Before discussing our aggregate computing approach to reconfiguration in Section 5, we motivate the investigation in terms of the following features of the paradigm.

Practicality. Aggregate computing comes with a set of *open-source tools* [18] including Domain-Specific Languages (DSLs), libraries, and simulators: the ScaFi language [19], its library, and the Alchemist simulator [20] will be leveraged in the experiments of Section 6.

Formal framework. Aggregate computing is based on field calculi, for which various formal results hold [18]. Prominent examples include *self-stabilisation* [24], the ability to build distributed algorithms guaranteed to eventually converge to *stable* (i.e. non-changing) outputs in finite time once inputs get stable; *space–time universality*, an expressiveness result that extends Turing-completeness to distributed computations, carrying the ability of expressing any effectively computable space–time function [18]; and *distribution independence*, the ability to express programs resilient to changes in system scale and density [18].

Macro/global perspective [17]. The field abstraction promotes reasoning in terms of collective behaviours, which facilitates *mapping desired global outcomes to local behaviour* [17,25].

Compositionality. By exploitation of the functional paradigm, language design, and libraries of self-organisation building blocks [24], it is possible to specify programs by *composing* blocks of collective behaviour.

Declarativity. The abstract execution and programming models provide flexibility and delay implementation decisions.

These features support reconfiguration in multiple ways. The availability of tools, both practical and formal, is useful for development and verification of reconfiguration logic in concrete systems. The macro perspective should help in designing reconfiguration policies that take into account non-local aspects (such as the resources available in large portions of the network, or connectivity patterns). Compositionality should support modular construction of reconfiguration policies. Declarativity, by allowing flexibility in the execution (e.g., where and how frequently) of reconfiguration policies, should help to address the heterogeneity of different deployments. This manuscript aims to substantiate these intuitive considerations.

3.2. Application partitioning and deployment via pulverisation

3.2.1. Core concepts

Pulverisation [11] is rooted in the idea that some *distributed application*, designed assuming a specific set of connected *logical devices* can be decomposed into smaller *components* (with clearly defined responsibilities and relationships) deployable on the underlying *infrastructure*, defined as a *network* of *hosts* (this terminology is also summarised in Table 1). Thus, when applicable (see next), pulverisation neatly separates the application logic from its target deployment infrastructure, allowing for the design of the former regardless of the latter. In other words, pulverisation can be seen as a way to separate functional and non-functional concerns, delaying the design of the deployment to later stages of the development (up to runtime). Similarly to other paradigms that also decouple business logic from deployment (such as task graphs [26] or reactive programming [27], where tasks or reactive operators can be assigned to different nodes of a network), pulverisation splits a collective behaviour into individual devices and their individual behaviours into uniform sets of components.

Concretely, pulverisation requires the single device logic to be separable into sub-components, which may require specific capabilities in order to be deployable on a host. Though the approach can be extended to arbitrary partitioning, we consider the classical partitioning in three main concerns:

1. *sensing*: acquiring information from the local environment;
2. *actuation*: acting on the local environment;
3. *business logic*: processing sensing data and taking decisions about actuation.

In some cases, the business logic can be further split into sub-components, e.g., to isolate the persistent *state* of the device, or to capture *communication* with other devices. The degree at which a specific application can be pulverised is ultimately determined by its design and technological framework: in some cases, the application may need to be modified to be pulverised, yet in other cases a neat separation is induced by the structural features of the paradigm/framework.

Notably, a prominent example of a "naturally pulverisable" paradigm is aggregate computing [11], which we introduced in Section 3.1. Thus, aggregate computing could be used simultaneously (i) as programming paradigm to specify the high-level collective behaviour of an application, and (ii) at the lower level, as a technology to program infrastructural reconfiguration policies. Some motivating factors that led us to propose this specific model for reconfiguration rules (cf. Section 3.1.3) also hold when deciding whether or not to adopt aggregate computing for the construction of the application: compared to other approaches amenable to pulverisation, such as Tuples-On-The-Air (TOTA) [28], the main reasons to lean towards aggregate computing are mainly related to the *practicality* (with relatively mature and available tools) and the *compositionality* (which permits easier reuse of existing collective behaviours)—cf. Section 3.1.3.

The problem of deployment of pulverised applications can be framed in the larger topic of *service placement* [29]. In particular, according to the service placement taxonomy in [29], our placement approach is *distributed*, *online*, and addresses *dynamic* and possibly *mobile* systems. Following the conceptual framework in [29], the pulverisation application model can be seen as a *connected graph of inter-dependent components*, but is actually more specific than that. The peculiarity of pulverisation is that it targets collective systems: a collective computation is split by their participating devices, and their individual behaviour is further split uniformly around a set of common component types.

3.2.2. Example

Consider, for instance, a simple system composed of two rain gauges controlling a valve, which should be open only if the average of the two gauges is above a threshold, or if either one is close to its maximum. The system can be naturally programmed considering three *logical devices*: the two gauges and the valve. Without pulverisation, the designer must also take a look on whether the system is actually composed of three *physical devices*, what they do support, and how they are connected. For instance, the valve might be attached to the same device controlling a gauge (so devices might actually be two); one or both gauges may be battery-powered *thin* devices (e.g., a LoraWAN mote), unable to host anything but minimal computation; there may or may not be the opportunity to send data to a remote server; the devices might be able to communicate directly or not, etc. All these aspects are irrelevant to the core business logic of the system, which can be easily defined without mentioning any of them, yet they need to be taken into

Table 1
Terminology. Parts of terms in parentheses may be omitted for brevity.

Term	Meaning
Infrastructure	A network of hosts (deployment domain)
(Logical) Device	An entity of the application business logic
(Physical) Host	Physical device that may host deployment units
Pulverised application/system	An application pulverised into devices and pulverisation components
(Pulverisation) Component	A deployment unit, part of a device functionality

account when designing the system with a classical approach, as such design encompasses lower-level details such as the actual communication among the involved *physical devices*.

Using pulverisation, the system behaviour should be specified to implement the core business logic as if the participants to the system were the three logical devices, considering them as if they were always available and connected. Once the behaviour is defined, it is broken down (pulverised) into small components. These components may specify a set of required capabilities that the physical hosts need to expose in order to host them; e.g., a component in charge of producing the gauge reading will need a sensor to be physically available, and a component evaluating whether the valve should open could need a better CPU than a microcontroller. Once done, the pulverised components can be deployed on the valid available physical devices, regardless of their actual hardware (as far as the aforementioned capabilities are provided) and network topology (as far as the network is not perpetually segmented). At this point, the same behaviour specification can be executed regardless of whether the gauges send readings via LoRa motes, the valve is or not attached to either of the gauges, the computation happens on the end devices, in the edge, or in the cloud, and so on.

3.2.3. On design and execution of pulverised systems

Pulverisation, at its core, is a technique for separation of concerns, separating application logic (written in a pulverisable programming paradigm) from distribution logic (as defined by the reconfiguration rules), coupled with automated distribution/partitioning of components (defined by the pulverisation middleware). Methodologically, applying pulverisation to a given application requires to [12]:

1. collect the functional requirements,
2. write the system behaviour specification, and
3. partition the application into pulverised components, to obtain what we call a *pulverised application* or *pulverised system*.

The pulverised components also work as *deployment units*, i.e., the minimally deployable software elements. A *deployment plan* is a mapping between deployment units to the deployment domain (the set of target hosts). In parallel to the partitioning, the available infrastructure can be analysed to build a model of the possible target infrastructures. At this point, several available deployment plans can be evaluated and compared, for instance via simulation, before performing the actual deployment of the pulverised components.

To sustain such approach, however, at deployment time, either a software maintainer or a *middleware* must perform a *wiring* of the deployment units in such a way that the system can operate on its underlying infrastructure. The *wiring* process can be supported by an optimisation-based approach, especially when the number of device is not too high, and we know in advance the topology of the network and the capabilities of the devices. We argue that, due to the openness of the ECC and the dynamicity of the system, it is not always possible to find an optimal solution. In this context, more scalable and effective approaches are needed, for example by leveraging a capability-based approach which can be exploited to constraint the placement of the components in the available hosts.

In case of *reconfiguration*, the deployment units may need to be relocated and a re-wiring performed—ideally automatically by the middleware. A middleware for managing pulverised systems has to deal with several concerns, thereby including scheduling, communication, and coordination. Of these, the most critical is the communication between the pulverised components, especially in a distributed setting like the ECC: realising communication channels among logical devices implies closing the gap between the *logical network* and the *physical network*. We consider the logical network as the ensemble of logical devices and their communication routes, which are defined *logically* when the application to be pulverised is designed. Later, as said, a logical device may end up being mapped to one or more physical devices (depending on whether its deployment units are hosted on the same physical device or not).

4. Middleware architecture

As discussed in the previous section, a pulverised application features two distinct levels of abstraction: the *logical* level, where the application is designed, and the *physical* level, where the application is deployed. To capture these abstractions, the middleware architecture is sliced in two distinct layers: the *Infrastructure Layer*, mapping the physical level, and the *Application Management Layer*, mapping the logical level. The former manages the aspects of the system closer to the metal, implementing the communication between hosts, which are network nodes (physical hosts, or, equivalently from the point of view of the system, virtual machines or containers). Also, it is responsible for managing the reconfiguration policies, which need to be aware of the physical network topology. The Application Management Layer builds instead on the network channels reified by the Infrastructure Layer and it is

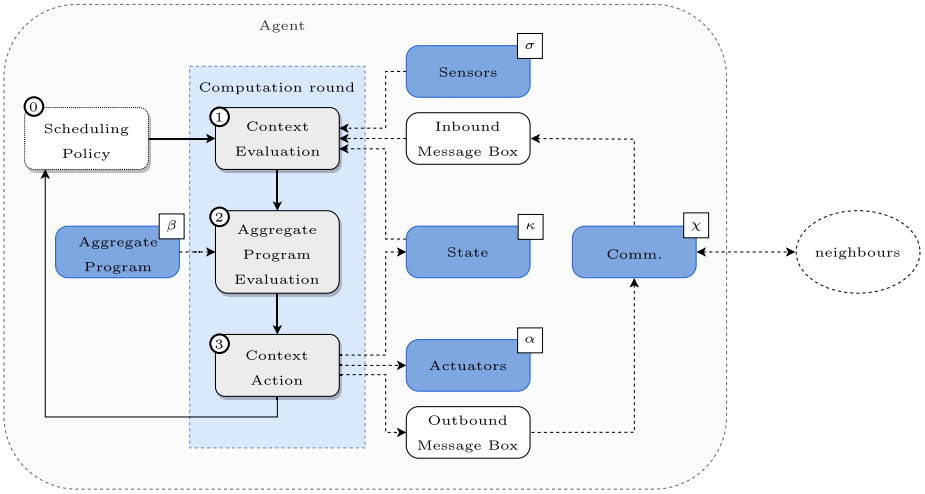

Fig. 1. Aggregate computing execution loop. The blue box represents the key components of the aggregate computing paradigm involved in the pulverisation model. In the pulverised view, each component is independent and managed by the pulverisation middleware.

designed to manage the pulverised components, moving them across the network as defined by the reconfiguration policies, and exposing virtually direct communication channels (whose actual shape, seen at the lower level, may be multi-hop) to the pulverised components that are being executed.

In other words, the Application Management Layer deals with the logical level of the system, reifying logical devices; while the Infrastructure Layer is responsible for managing the physical level of the system, thus including hosts.

In a typical workflow, the developer willing to operate on such middleware implements the components of each logical device conforming to the interface provided by the middleware, which must provide means to apply the partitioning as defined by the pulverisation approach. Then, the developer registers them to the middleware via a dedicated API, letting the middleware manage the deployment units. Similarly, the developer can implement reconfiguration policies by registering them in the *Reconfiguration Manager* via a dedicated API. Once the components are registered and the middleware is configured, the developer is responsible for deploy the middleware instance on the infrastructure, i.e., via an executable file or a container.

In the following discussion, we detail how these layers work.

4.1. Pulverisation and aggregate computing integration

The pulverisation model is strictly related to the Aggregate Computing paradigm, since it provides a (natural) way to partition a *logical device* into independent components. This partition is devised to preserve the computational model of Aggregate Computing, where the computation is performed in a distributed fashion by the devices of the system.

In Aggregate Computing, each device executes a program in a round-based fashion, where the computation is divided into three main phases:

1. *Context Evaluation*: the device acquires the sensors' values, the previous state, and the messages from the neighbours;
2. *Program Evaluation*: from the data acquired in the previous phase, the device computes the program producing the output;
3. *Context Action*: from the output of the program, a new state is produced, as well as a new message for the neighbours, and the actuators are activated based on the prescriptive actions.

The aforementioned phases are reported in Fig. 1, where the Aggregate Computing execution loop is depicted. Moreover, The blue box represents the Aggregate Computing components involved in the pulverisation partitioning, where each of them is managed by the middleware architecture (cf. Section 4.2).

In this context, the pulverisation model shifts the focus from a single device execution, to a more distributed model where the components can be deployed on different hosts, and is the responsibility of the middleware to manage the communication among them to ensure the round-based execution correctness.

4.2. Application management layer

The Application Management Layer is responsible for reify logical devices and hence execute one or more pulverised applications. More in detail, this layer manages the *deployment units*: a deployment unit is a deployable software package that consists of a possibly dynamic collection of one or more pulverised components belonging to the same logical device instance. The deployment unit can change by hosting a new component or removing an existing one, based on the current deployment plan of the system.

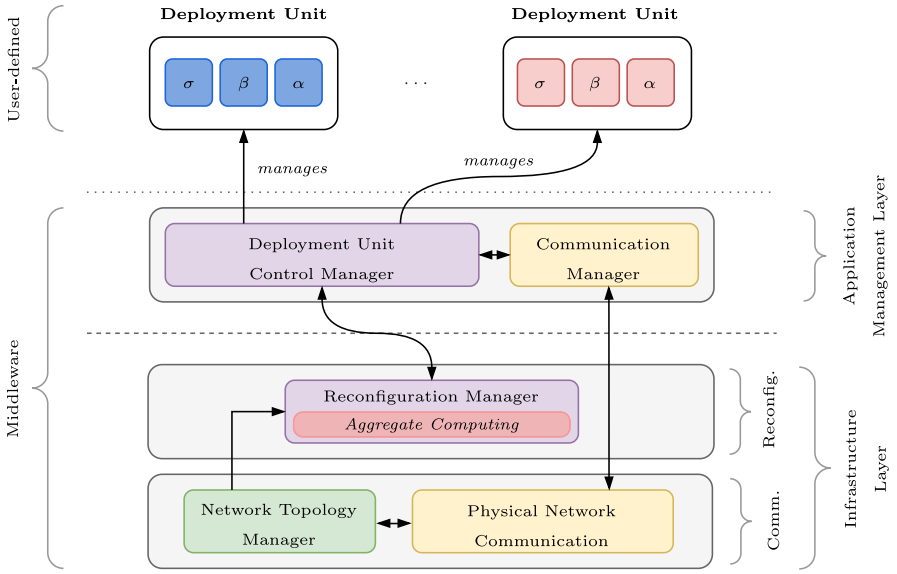

Fig. 2. Representation of the two middleware layers and the mapping between them in the architectural model. In the *Application Management Layer*, a link between components represents logical communication. In the *Infrastructure Layer*, a link between hosts represents an existing direct network communication channel. The *Aggregate Computing* subcomponent in the *Reconfiguration Manager* can be used to manage global reconfiguration policies.

The management of the deployment units is performed by the `Deployment Unit Control Manager`. The information about the actual system's configuration is provided by the *Infrastructure Layer*, and it is used by the *Application Management Layer* to manage the deployment units accordingly.

This is the layer exposed to the application developer, since it provides a logical abstraction of the system, without leaking details of the underlying physical infrastructure. In fact, in this layer, the developer deals with logical devices and components associated with them, without worrying about *where* the components will be executed or *how* they will communicate with each other, thus abstracting away details on how the communication between the pulverised components will be performed in practice. In fact, a different component called `Communication Manager` manages the communication among pulverised components at runtime, based on the information provided by the `Physical Network Communication`, which is part of the *Infrastructure Layer* and has visibility of hosts, it can thus route and retrieve messages among them.

Fig. 2 shows the aforementioned layer and the interaction between the `Deployment Unit Control Manager`, and the `Communication Manager`.

4.3. Infrastructure layer

The *Infrastructure Layer* is responsible for managing hosts and the communication among them. In particular, this layer can be further divided into two sub-layers: *Reconfiguration Layer* and *Communication Layer*. The former is responsible for managing the reconfiguration of the deployment units according to the adopted reconfiguration strategy, while the latter is responsible for realising the actual communication among the hosts and keeping track of the physical network topology (which hosts can directly communicate with which, and how messages should be routed so that a host can reach any other one).

From Fig. 2, the `Reconfiguration Manager` communicates with `Deployment Unit Control Manager` of the *Application Management Layer* to perform runtime reconfigurations of the system. In fact, the `Reconfiguration Manager` exposes, for each application executed in the middleware, which deployment units are currently selected to be executed on the local host. This information is used by the `Deployment Unit Control Manager` to modify the deployment units, starting and stopping the affected components as described in Section 4.2.

The `Reconfiguration Manager` must be aware of the current infrastructure topology to be able to select which deployment units should be available and where. Thus, it can access such information from the `Network Topology Manager`, the middleware component dedicated to keeping an up-to-date view of the current network topology. The information provided by the `Network Topology Manager` can be (and typically is) enriched with data such as channel latency, jitter, and status information at the destination (for instance, average CPU load, memory usage, etc.). All these additional information can be exploited in the reconfiguration policies to take more informed decisions. For instance, when a latency-sensitive reconfiguration policy is adopted, the latency information can be accessed by the `Reconfiguration Manager` to identify the best host in which to deploy a specific deployment unit, minimising latency. Similarly, a latency-based global policy can use this additional information to place the communication components into physically close hosts, reducing the overall latency of the communication for the devices in that

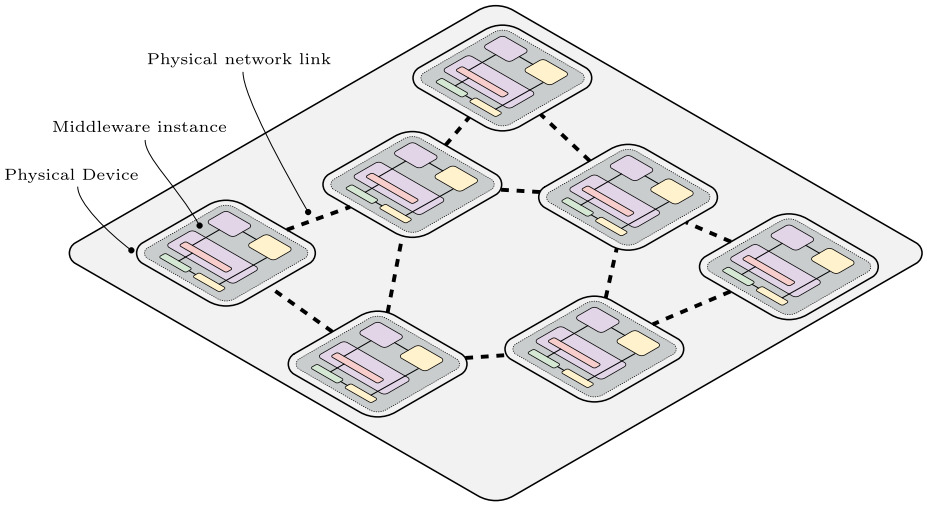

Fig. 3. Representation of a possible deployment of the middleware showing the *Reconfiguration Manager* component supported by the *Aggregate Computing* for managing the reconfiguration policies. The dashed lines represent the physical network communication channels between the devices, reflecting the physical neighbourhood relationships. The gray rounded rectangle inside each device represents the middleware instance running on the device. The coloured rounded rectangles (inside the gray one) represent the middleware's components, as depicted in Fig. 2.

physical area. A more detailed description on how the reconfiguration policies can be implemented and managed by the middleware is provided in Section 5.

Finally, the `Physical Network Communication` is responsible for performing communication among hosts through any established network protocol. The role of this component is twofold: enabling the communication between pulverised components hosted on different network nodes (supporting the *Communication Manager*), and providing the `Network Topology Manager` with information required to keep an updated infrastructure map (for instance, by sending periodical heartbeats and/or status updates).

In Fig. 3, we depict several hosts (thick squares) running the middleware instances (coloured rounded rectangles), connected to each other by physical network communication channels (dashed lines). The `Reconfiguration Manager`, as reported in Fig. 2, can leverage the *Aggregate Computing* paradigm to manage the reconfiguration policies. In this context, the figure shows that the *Aggregate Computing* part is executed on each host, and via the *Physical Network Communication* component, the coordination messages are exchanged among the neighbour hosts of the physical network. From the figure emerges the complete distributed nature of the middleware, where each host runs an instance of it and communicates with the other instances in a neighbour-to-neighbour fashion.

In an ECC infrastructure, heterogeneity can lead to the adoption of different communication protocols across different hosts. As an example, LoRaWAN motes may communicate with the homonym protocol with a LoRa gateway, which then sends the data to an edge server via MQTT, and it, in turn, communicates via HTTPS with some cloud-hosted instances. In such a scenario, the `Physical Network Communication`, provides a common interface to interact exposing primitive operations, abstracting away the actual communication protocols used by the hosts.

The overall mapping between the logical devices and their physical hosts is depicted in Fig. 4, showing an example of how the proposed architecture is capable of enforcing deep decoupling between logic devices and their perceived network and hosts executing them, connected to a physical network.

5. Aggregate computing for dynamic reconfiguration

Although pulverisation is an effective way to flexibly deploy Aggregate Computing applications on the ECC, the latter can also be exploited to implement the reconfiguration policies, especially in a highly dynamic environment like the ECC. In this section, we show that Aggregate Computing is a practical tool to express reconfiguration policies in a high-level fashion, from a global stance (positively answering RQ2). This way the role of Aggregate Computing is potentially dual: on one side, it provides a suitable programming model for implementing collective behaviours, on the other side, it can be used to implement global reconfiguration policies.

5.1. Syntax introduction and aggregate operators

We introduce the syntax adopted in the following examples. All the code snippets are written in Scala 2, which is also the language used in the ScaFi framework [19].

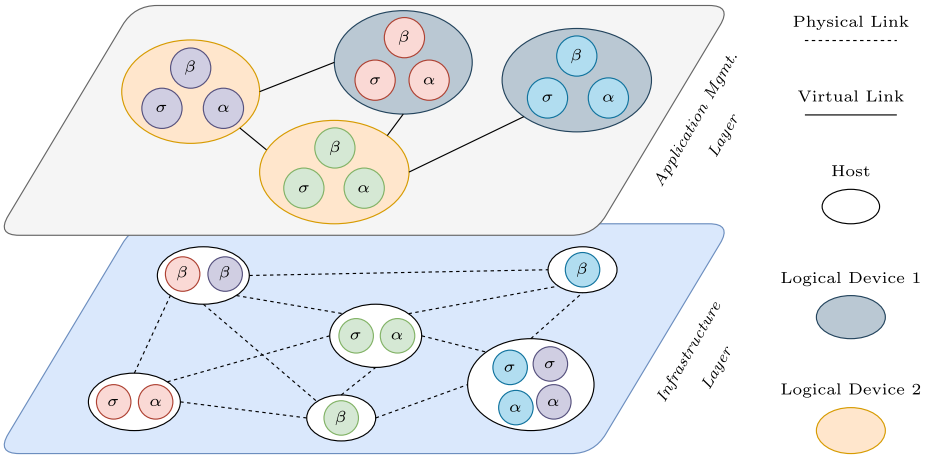

Fig. 4. Representation of a simple system composed of four logical devices (Application Management Layer), each pulverised in three components. The figure shows the mapping between the logical devices and the hosts (Infrastructure Layer). Note how the pulverisation into components and the creation of deployment units enables a high degree of decoupling between logical devices and hosts.

5.1.1. Scala syntax

To simplify the understanding of the code snippets for readers unaquainted with Scala 2, we introduce the syntactic elements used in this paper's examples. The reader is referred to the Scala 2 language specification[1] for a more comprehensive overview of the language.

A function is defined by the `def` keyword, followed by the function name and, within parentheses, the function input parameters, whose type is annotated after a colon; finally, the return type annotation concludes the function signature. Scala supports currying, namely, multiple parameter lists (each enclosed in parentheses) can be defined. Additionally, generic functions may feature type parameters enclosed within square brackets between the function name and the parameter list(s) (for instance, `[A]` indicates that the function is generic in `A`).

Anonymous functions (lambda expressions) can be defined with the `(parameters) => body` syntax. where `parameters` is the list of input parameters and `body` is the body of the function. Lambda expressions with a single parameter can be written without parentheses around the parameter list. Additionally, the parameter list and the `=>` symbol can be omitted if the body uses the parameters only once: for instance, `_+1` is equivalent to `x=>x+1`, and `_+_` is equivalent to `(x,y)=>x+y`.

5.1.2. ScaFi syntax and aggregate operators

The ScaFi framework provides a set of primitives that provide a practical implementation of the higher-order field calculus [30] and can be combined to realise complex aggregate programs. A description of these operators follows.

Neighbouring — `nbr`. This operator has a dual function: the data it is fed with is shared with the neighbours, and returns a data structure where each value is the one shared by the corresponding neighbour, thus implementing a sort of send/receive operator that enables operations in *space*. For instance, `nbr(0)` shares with the neighbours the value 0; generally, any expression result can be used as parameter of `nbr`.

Repeating — `rep`. The repeating operator captures state evolution. It requires an initial value and a function that computes the next state based on the previous one, enabling stateful operations in *time*. For instance, `rep(0)(x=>x+1)` will produce a sequence of integers starting from 0.

Domain separation — `branch`. The domain separation is the distributed equivalent of branching in concentrated systems. Devices which share the same condition will compute the same branch of the program, and will not communicate with the devices computing the other branch, thus de-facto partitioning the network. Note that, within aggregate programs, the native Scala `if` statement is not allowed, as it interferes with the distributed branching semantics.

Functional selection — `mux`. Given the behaviour of the `branch` operator, an additional form of branching is required for computations that need to communicate with all neighbours on both branches and return only one of the two results. This abstraction is provided by the `mux` operator. For instance, `mux(cond){trueBranch}{falseBranch}` will compute the `trueBranch` if `cond` is true. `falseBranch` otherwise.

[1] https://scala-lang.org/files/archive/spec/2.13/.

Folding — `foldHood` and `foldHoodPlus`. Since `nbr` produces a value per neighbour, its usage is only meaningful if the values are combined, or, in a functional programming sense, *folded* into a single value. The `foldHood` operator provides this functionality. It takes as input an initial value, a reduction function, and an expression containing a neighbouring operation. The operation is executed, and the resulting field, mapping device identifiers to values, is then folded into a single value by applying the reduction function element by element from the provided initial value. `foldHoodPlus` is similar to `foldHood`, but discards the local field value. For instance, `foldHood(0)(_+_)(nbr(1))` sums the values shared by the neighbours and the local value, while `foldHoodPlus(0)(_+_)(nbr(1))` sums the values shared by the neighbours excluding the local value.

5.1.3. Aggregate building blocks

Together, the aggregate operators are universal, in the sense that they can be used to implement any distributed algorithm. However, many of these may not be *self-stabilising*, i.e., they may not converge to the correct behaviour after any disruptive event [31]. To simplify the construction of self-stabilising software, a set of common patterns have been identified and implemented as building blocks in the ScaFi framework [24]. They implement common collective behaviours in a self-stabilising fashion, and, crucially, their functional composition produces self-stabilising algorithms. We briefly introduce them in the remainder of this section.

Gradient-cast — G. This building block propagates information from the closest source to its surroundings (possibly, the whole network), by building a gradient based on the provided distance metric.

Converge-cast — C. The converge-cast is the dual of the gradient-cast. It collects information from the surroundings towards a sink, based on a potential field, often built using the `G` operator: it is common for the result of a `G` operation to be fed as parameter of a call to `C`.

Sparse-choice — S. The sparse-choice building block implements symmetry breaking in homogeneous networks, it takes a `grain` parameter and a distance metric, and partitions the network into areas whose size is proportional to the `grain` parameter.

5.2. Aggregate-based reconfiguration policies

A reconfiguration policy is a function outputting a `Set[Component]`, representing the components that must be executed by the local host. All the information required to manage the reconfiguration are accessible via sensing operation provided by the Aggregate Computing framework. From a global stance, the aggregate function running using the ECC as a single computation device outputs a field that associates to each point a set of pulverised components. Of course, provided that the types of the *pulverised* components are compatible with the local host, approaches other than aggregate computing could be used; however, we will show in this section that adopting the latter provides guarantees and allows to succinctly express rich policies. We will do so by presenting a series of increasingly complex examples, to showcase how the approach can scale easily even when policies become intricate. For simplicity, we assume three pulverised components: `Behaviour`, `Sensor`, and `Actuator`.

In Listing 1, we present one of the simplest possible policies: all the components local to the device.

```
1 def main(): Set[Component] = Set(Behaviour, Sensors, Actuators)
```

Listing 1: All components are local to the device.

The idea of using Aggregate Computing to implement reconfiguration policies is rooted on the idea that networked hosts represent samples (points) over a manifold in which multiple metrics can be defined. Although designed to support spatial and spatio-temporal computations, in principle Aggregate Computing can execute on any Riemannian manifold, as far as suitable metrics are defined (many algorithms assume the strict triangle inequality to hold). As an example, a valid metric could be the round-trip latency between two directly communicating hosts, or the sum of the CPU usage: in the former case, hosts connected by low-latency links will see each other closer than those with high-latency links; in the latter case, hosts with high CPU usage will be perceived as far from the others. Listing 2 shows a policy that offloads the behaviour component to the nearest host with the lowest CPU load and latency below a threshold. It does so by building a field of latencies, and consider only the directions in which it is below a threshold (Line 4 to Line 6), then selecting among them the host with the lowest CPU load (Line 8), and finally propagating its decision (Line 10 to Line 13). More specifically, the `foldHoodPlus` function is used to collect the latencies and CPU loads of the neighbours, and filter out the neighbours with a latency above the threshold. Of the remaining neighbours, we are interested in the one with the lowest CPU load, since we want to offload the behaviour component to the host with the lowest CPU load. This operation is trivially performed by the Scala standard library function `minBy`, which selects the minimum element of a collection based on a specific criterion (in this case, the CPU load). Once the candidate host is selected, the decision is propagated to the neighbours, via the `nbr` operator by providing the id of the selected host and the component to offload — in this case, the `Behaviour` component. Finally, the `foldHoodPlus` function is used to collect the decisions of the neighbours, by accumulating only the components that are meant to be executed locally. However, note that this policy has several flaws: first, it never propagates information but to the immediate surroundings of each host; second (consequence of first), the policy may thus be unstable and create cycles in which nodes perpetually offload and get back their own behaviour component. In other words, this solution is *non-self-stabilising*: when using Aggregate Computing to coordinate large scale systems, it is indeed better to avoid low-level mechanisms (`nbr` and

rep), and rely on high-level building blocks. These building blocks are built in such a way that their composition guarantees the self-stabilisation property, as proved in [32].

```
def maxLatency = ... // the maximum admitted latency
def cpuLoad = ... // the local cpu load
def main(): Set[Component] = {
  val selected = foldHoodPlus(List())((acc, (lat, c)) =>
    mux(lat < maxLatency){ acc :+ (mid(), (lat, c)) }{ acc }
  )(nbrLatency() -> nbr(cpuLoad))
  // Select the node with lowest CPU load
  val (bestId, _) = selected.minBy { case (_, (lat, cpuLoad)) => cpuLoad }
  // Propoagate the offloading decision
  val decision = nbr { (bestId, Behaviour) }
  foldHoodPlus(Set())((acc, (id, comp)) =>
    mux(id == mid()){acc + comp}{acc}
  )(decision) ++ Set(Sensor, Actuator)
}
```

Listing 2: Rule expressing the intent to offload the behaviour component to the nearest host with the lowest CPU load and latency below a threshold.

Thus, as a third example, we show in Listing 3 a policy in which all the behaviour components get centralised into the node with the best available CPU using functions that operate over building blocks, and thus guarantee self-stabilisation. To do so, it builds a field of tuples containing node id and CPU capacity, of which the maximum in the network is selected via `gossip` (Line 4), then, from the node with the best CPU, we propagate a potential field (Line 5), which is used, in turn, to build a spanning tree over which all behaviours are collected (Line 6). Finally, every host decides which component(s) to run: `Sensor` and `Actuator` are locally executed (if available), and the `Behaviour` is offloaded to the node having the lower CPU load on the network (Line 9).

In this example, the `gossip` function is used to propagate in all the network the maximum CPU capacity and the id of the node having it. Then, based on the id of the node with the best CPU capacity, a gradient is propagated having as source the node with the best CPU capacity. As stated above, this gradient is used to backtracking the potential field generated by the `gradient` function, and collect into the source node (the one with the best CPU capacity) all the intended components to be executed. This operation is managed by the `C` function, which is used to collect the components to be executed by the source node. Finally, the `flatMap` function is used to select the components to be executed by the local node.

```
// local CPU capacity, e.g., IPC * frequency * cores
def cpuCapacity = ...
def main(): Set[Component] = {
  val (id, bestCpu) = gossip(cpuCapacity, max)
  val potential: Double = gradient(source = id == mid())
  val selected = C(potential, _ ++ _, Set((mid(), Behaviour)), Set())
  selected.flatMap { case (id, comp) =>
   if (mid() == id) Set(comp) else Set()
  } ++ Set(Sensor, Actuator)
}
```

Listing 3: Rule expressing the intent to offload the behaviour to the best CPU in the system.

Of course, centralise everything is not a policy that can scale with the number of nodes in the system. In Listing 4, the last snippet of this short tour, we show how to achieve hybrid coordination through the SCR pattern [33]. We first split the whole ECC into regions of coordination, each one with a leader elected through `boundedElection` [34] (Line 5). We then propagate a potential field from each leader (Line 6), preferring low-latency connections, which is used to build a regional spanning tree that accumulates information into the partition leader (Line 7). Finally, we select the components to run locally, with local leaders taking charge of all the regional behaviours (Line 8 to Line 10).

The `boundedElection` function is used to elect the leader of the region based on the CPU capacity of the nodes. With this function, a number of leaders are elected forming regions that are bounded to `maxLag` latency, respecting the `limit` parameter. Then, for each leader, a potential field is propagated to determining the potential of the region. As for the previous example, via the `C` function, the components to be executed are collected into the leader node.

6. Evaluation

In this section, we exercise an implementation of our proposed middleware and reconfiguration programming approach. In particular, we show how the SCR pattern can be exploited to manage the dynamic relocation of the component's execution at runtime according to the pulverisation partitioning.

```
// local CPU capacity, e.g., IPC * frequency * cores
def cpuCapacity = ...
def maxLag = ... // the maximum admitted latency
def main(): Set[Component] = {
  val leader = boundedElection(strength=cpuCapacity,metric=nbrLag, limit=maxLag)
  val potential: Double = gradient(source=leader == mid(), metric=nbrLag)
  val selected = C(potential, _ ++ _, Set((mid(), Behaviour)), Set())
  selected.flatMap { case (id, comp) =>
    if (mid() == id) Set(comp) else Set()
  } ++ Set(Sensor, Actuator)
}
```

Listing 4: Rule expressing the intent to offload the behaviour to the node with best CPU capacity within a bounded latency.

The discussion is organised as follows: Section 6.1 introduces the goals of the evaluation and how they provide evidence for the research questions; Section 6.2 describes the case study; Section 6.3 defines the methodology and provides detail on the reconfiguration strategies under test; finally, in Section 6.4 we show the final results.

6.1. Evaluation goals

The goal of the evaluation is to provide evidence that:

1. the host allocation of the pulverised components of a distributed application can be modified at runtime using the proposed architecture, more specifically (cf. RQ1);
2. it is possible to implement a distributed reconfiguration from a global stance, specifically by using the Aggregate Computing paradigm (cf. RQ2);
3. the ability to reconfigure the system at runtime is helpful to achieve better QoS and performance trade-offs compared to a traditional pulverised deployment with no runtime reconfiguration (cf. RQ3).

6.2. Scenario

We consider a network of one thousand heterogeneous hosts on which multiple distributed applications are executing concurrently, among which our pulverised application. Hosts can be either *thin* or *thick*, depending on their computational capabilities. Thin hosts have limited computational capabilities (e.g., due to being battery powered), but they are the majority of the devices ($^{950}/_{1000}$), and are equipped with sensors and actuators that are required for the application—they could be, for instance, smartphones. Thick hosts, instead, have more computational resources and memory, but they are fewer (only $^{50}/_{1000}$) and do not have sensors or actuators—they could be, for instance, edge servers or cloud instances. For simplicity, we assume that the *thick devices* share the same hardware specifications, since the main focus is on evaluate the feasibility of reconfiguration approach, and not on accurately modelling the hardware.

Once pulverised, our application is composed of multiple logical devices, each of which is made of three components: *sensors*, *actuators*, and *behaviour*. Sensors and actuators require specialised hardware to be executed, and must thus remain aboard thin hosts, while the behaviour component can be executed on either host type. Even though thin hosts could in principle run the behaviour component, they always try to offload it to a thick host in order to save battery. To observe the system under stress, we assume that, when the behaviour component is executed on a thick host, it drains 3% of its computational capabilities.

With the proposed reconfiguration approach, we aim to show that a more efficient relocation of the behaviour component can be achieved, compared to a local counterpart where each device tries to offload its behaviour to the closed thick host. In this last scenario, if a thick device became overloaded, the pre-defined thin devices requiring the offloading would not be able to offload their behaviour component, thus becoming non-operational. In this context, the proposed approach tries to better adapt the behaviour component allocation by shaping the regions of coordination according to the thick devices' load, thus dynamically selecting the best thick device to host the behaviour component, namely the one with the lowest CPU load (if possible). Note how this condition would make a static pulverisation approach hardly viable, as the overall computational capacity of the thick hosts is not large enough to host the behaviour of all the logic devices: the implementer would need to identify a set of thin hosts to "sacrifice", as they would quickly run off battery. In our testbed we show that, instead, a dynamically reconfigurable system is viable albeit, of course, with non-perfect QoS as it can be configured to share the burden among several thin hosts.

6.3. Experimental configuration

6.3.1. System dynamics

Every experiment simulates the network for 12 min (720 s). We perform two experiments. In the first one, we are interested in observing how the system reacts to changes in the available thick host capacity, thus verifying whether or not it can adjust in

response to changes in the system dynamics even though the network topology is unchanged. To do so, we apply a dynamic load driver to the thick host, simulating a variable load from the applications that are being executed concurrently with the pulverised one. In the second experiment, we stress the reconfiguration capabilities in case of graceful degradation and slow recovery. To do so, we turn off batches of 25% of all thick hosts (simulating, e.g., a brown out) until the system is left with only 25% of its original capacity. We then proceed to a progressive reactivation. Crucially, in these cases the network is allowed to be segmented, thus some thin hosts may be forced to execute the behaviour component locally, as no thick host is reachable. Our goal is to observe whether the system can remain functional in case of disruptions affecting the network topology, and whether it recovers to its original QoS once the system recovers from the disaster.

6.3.2. Network topologies

The devices are displaced on the network according to two different network topologies: a Barabasi–Albert network [35]; and a Lobster network [36] with at most 4 hops from the backbone and maximum node degree limited to 20 direct neighbours. The former implements a scale-free network, while the latter is representative of a backbone network composed of a set of edge-server and multiple devices connected to them forming small sub-networks. In both cases, we mark as thick hosts the 50 nodes with the highest degree in the network.

6.3.3. Metrics

In all experiments, we measure the QoS of the system through a metric that captures the fraction of thin hosts that can successfully offload their behaviour pulverised component to a thick host. In this context, we intend QoS as an indicator of the system's ability to adapt to the changing conditions of the system, and consequently of the reliability and availability of the system. Thus, by denoting the set of thin hosts as τ and the set of thin hosts whose behaviour has been successfully offloaded to a thick hosts as $\tau_o \subseteq \tau$, then $QoS = \tau_o/\tau$. Thus defined, the QoS is a value in the range $[0, 1]$ representing the probability that a thin host successfully offload its behaviour component to a thick host. When $QoS = 1$, all thin hosts have successfully offloaded their behaviour component to a thick host, while when $QoS = 0$ no thin host has been able to offload its behaviour component.

6.3.4. Baseline

The baseline we compare with is a system where offloading is pre-defined: every thin host offloads its behaviour component to the nearest thick host (measured in terms of hop count). In case multiple thick hosts have the same hop distance from the thin host, then the one with the shortest latency (round-trip time, measured at the beginning of the experiment) is selected. Note that, with a no reconfiguration system in place, the result of $QoS \neq 1$ would imply that a certain percentage of devices is not operational. The main reason for this is that the thick hosts are overloaded, and they cannot host all the behaviour components of the thin devices requiring the offloading. The thin devices not capable of offloading their behaviour because of this condition are not operational. With support for reconfiguration at runtime, the system could keep operating, but with a degraded QoS. In our experiments, we implement the baseline strategy using the same reconfiguration framework in order to get a fair comparison, and focus on the benefit of an aggregate specification.

6.3.5. Dynamic reconfiguration

We showcase the viability and efficacy of the proposed approach by realising a collective reconfiguration strategy that, in few lines of code, captures a complex collective behaviour.

```
1  def isThick: Boolean = ... // true if the device is thick
2  def offloadWeight: Int = ... // the offload weight
3  def cpuCost(): Int = if (isThick) /* cpu load */ else offloadWeight
4  def behaviour = ... // the local behaviour component
5  def main(): Set[Component] = {
6    val potential = G(source=isThick, field=cpuCost(), acc=_+_, metric=cpuCost)
7    val managed: Set[Behaviour] = C(potential, _++_, Set(behaviour))
8    var load = cpuCost()
9    val runOnThick: Set[Behaviour] =
10     if (isThick) managed.takeWhile{load+=offloadWeight; load<100}
11     else Set()
12   val onLeader = G(isThick, runOnThick, identity(), cpuCost)
13   if (isThick) onLeader
14   else Set(behaviour) -- onLeader ++ Set(Sensor, Actuator)
15 }
```

Listing 5: SCR-based dynamic reconfiguration.

We let each thick host act as a source of a potential field whose lowest value, centred in the thick host itself, is its current CPU load. This field grows at every hop based on the expected offloading cost (in this case, 3% of the overall CPU usage). In case a node is reached by multiple gradients, the one with the lowest value is selected, creating a natural partitioning of the network. All these operations are conveniently summarised by the `G` building block at line 6 of Listing 5. Interestingly, since the lowest value of the potential field is associated with the available CPU of the thick hosts, hosts with a higher CPU load will have a smaller

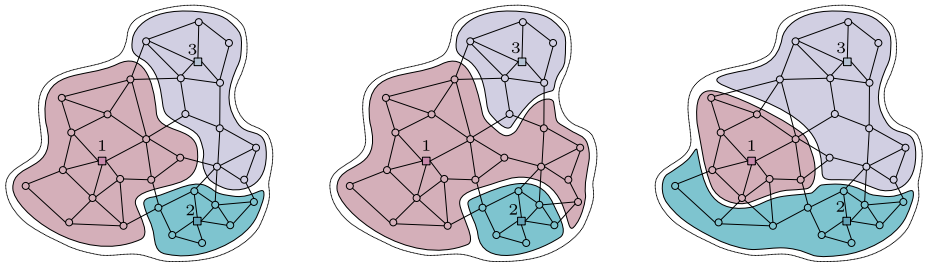

Fig. 5. Visual representation of three changing conditions of the reconfiguration algorithm. The coloured shapes represent the regions coordinated by the leaders (squares inside each region). The region's area is proportional to the leader's CPU load.

region of influence, and thus the system will automatically perform a load balancing operation. Now, the potential can be used to accumulate all behaviours to be executed on the thick host who is leading a certain area, again supported by an existing building block: `C` (Listing 5, line 7). The thick host will select the behaviours to be executed locally (Listing 5, lines 8—11) and propagate the decision along the potential (Listing 5, line 12). Finally, every device will return the set of components to be executed locally: the selected ones for the leading thick host; `Sensor` and `Actuator` for the thin ones plus the local behaviour if not selected for the offloading (Listing 5, line 14). Overall, this strategy is an implementation of the SCR pattern, which can be realised in a handful lines of code.

To better understand the reconfiguration algorithm described above, we provide a visual representation of the algorithm in Fig. 5. Initially, the regions are formed via SCR based on the actual CPU load of leaders (left side of the figure), depicted in the figure as squares. The middle part of the figure shows how the regions are dynamically adapted when leaders 3 and 4 are increasing their CPU load. When this condition occurs, the regions of leaders 3 and 4 are reduced, and the regions of leader 1 is increased to accommodate the thin devices that were previously in the regions of leaders 3 and 4. Finally, the right side of the figure shows the opposite condition, where leaders 3 and 4 are decreasing their CPU load, and the regions of leaders 3 and 4 are increased, while the region of leader 1 is reduced. With this dynamic reshaping of the regions, the system can adapt to the dynamic changing conditions of the system, and maximise the number of thin devices that can offload their behaviour component.

To keep the model simple, we consider only the CPU load of each host in the System. This is mainly due to the fact that we suppose the thick device share the same hardware characteristics, so even though additional parameters (e.g. CPU capacity) could be considered, in this simulated scenario they do not provide any additional values for the purpose of this evaluation. However, more complex metrics based on the combination of multiple parameters can be defined, and such investigation represents a promising direction for future works.

6.3.6. Tools and reproducibility

We repeat every experiment for 10 times with a different random seed; the results presented in this manuscript show the average $\pm$ one standard deviation (1σ). The experiments have been executed using the Alchemist Simulator [20]. The Reconfiguration specifications have been written in Scafi [19]. Data analysis and visualisation were performed using xarray [37] and matplotlib [38] respectively. For inspectability and reproducibility, the experiments have been released with a permissive open-source licence[2] and archived for future reference on Zenodo [14]. For the sake of brevity, we only show the results of the most relevant experiments. In the aforementioned repository, we provide the complete set of 72 charts, including experiments performed with a different device count and behaviour CPU load.

6.4. Results

Results are depicted in Fig. 6 for the dynamic load experiments, and in Fig. 7 for the graceful degradation and recovery. In both cases, the collective strategy shows better overall results in terms of QoS when compared to the local one (baseline) after a period of adaptation and stabilisation of the distributed data structures, during which the results may get worse. In fact, the aggregate policies require some time for the self-stabilisation process to complete, and until stabilisation is reached, the QoS can be affected negatively. Moreover, we observe that the collective strategy has a more pronounced sensitivity to network density and segmentation: the algorithm is naturally capable of finding alternative offloading routes and exploit them, but doing so is faster as the network is denser, and it is impossible to do in case of segmentation. In fact, in the graceful degradation and recovery scenario, if we compare the results of the two different network topologies, we observe that the proposed approach (although still better performing than the baseline) obtains much better results in the scale-free network. The reason is that in the lobster topology segmentation as a consequence of the random removal of a node is much more likely compared to a scale-free topology.

Similarly, in the variable load experiment we observe almost the same performance on a lobster topology, and yet a considerable improvement in the scale-free topology. The reason behind this behaviour is due to the network density (the ratio between the actual number of edges and the maximum possible number of edges in the network): denser networks have more alternative paths to choose from, hence allowing the proposed adaptation to select from a wider spectrum of alternatives.

[2] https://github.com/nicolasfara/reconfiguration-experiments.

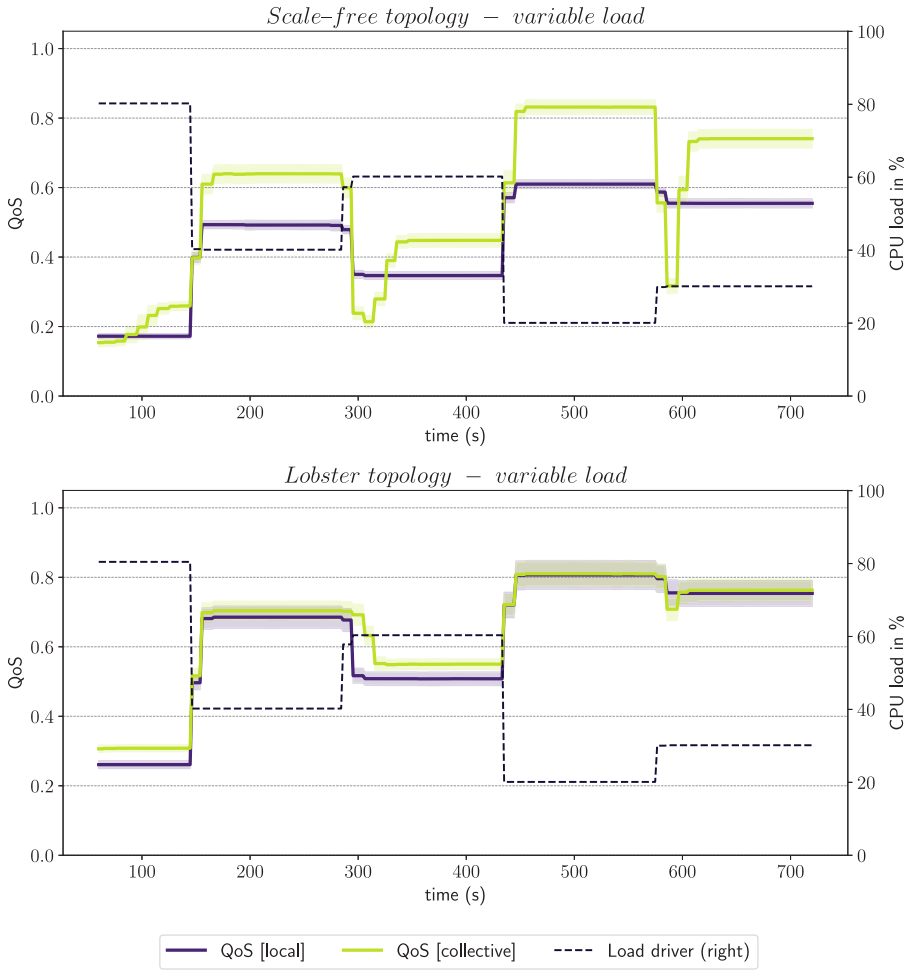

Fig. 6. Results for the dynamic load scenario with a scale-free (upper) and lobster (lower) topology. The collective strategy is depicted in green, the local one in violet, and the load driver is dashed and black. Shadows around the lines represent $\pm 1\sigma$. The collective strategy shows better overall results in terms of QoS, at the cost of longer transients due to adaptation and stabilisation of the distributed data structures. Also, the collective strategy benefits from denser networks: the more alternative paths are available to choose from and the lower the probability of segmentation is, the better the overall performance.

6.5. Threats to validity

Our experiments do not consider a realistic energy use model. Frequent system reconfiguration, even though leading to a better QoS, may cause the overall power usage to be greater than a statically defined system. However, this ultimately boils down to the construction of an accurate energy model and subsequent selection of an appropriate reconfiguration policy, which are out of the scope of this work.

Related to this point is the actual heterogeneity of the hosts on the ECC. In our experiments, we simplify by talking about thin and thick hosts, but in reality the situation is way more nuanced: smartphones and wearables come in all sorts of hardware configurations, and, similarly, the infrastructure is very diverse (both in the edge and in the cloud). Capturing the heterogeneity with the proposed approach implies a non-trivial effort describing appropriate metrics that, in turn, can drive the reconfiguration policies, similarly to what has been discussed for the energy model.

Different network topologies may have shown different results. In our experiments, we tried to cover two very different topologies in the attempt to stress the system under very different conditions, namely a scale-free network and a lobster network. Although those topologies can approximate complex infrastructures like the ECC, further experiments are needed using real-world data coming from real-world infrastructures.

Although we measure degradation in terms of failures of the thick hosts, we did not consider the failure of the thin hosts. We believe that this is more representative of a true stress condition, as the failure of thin hosts would have *reduced* the overall load of the system, as every thin host maps onto a pulverised device.

Another aspect we did not consider is the latency dynamicity. In these experiments, we fix a channel latency value for each link in the network, and we simulate using that value. Also, we assume no packet loss (or, equivalently, that package retransmission is

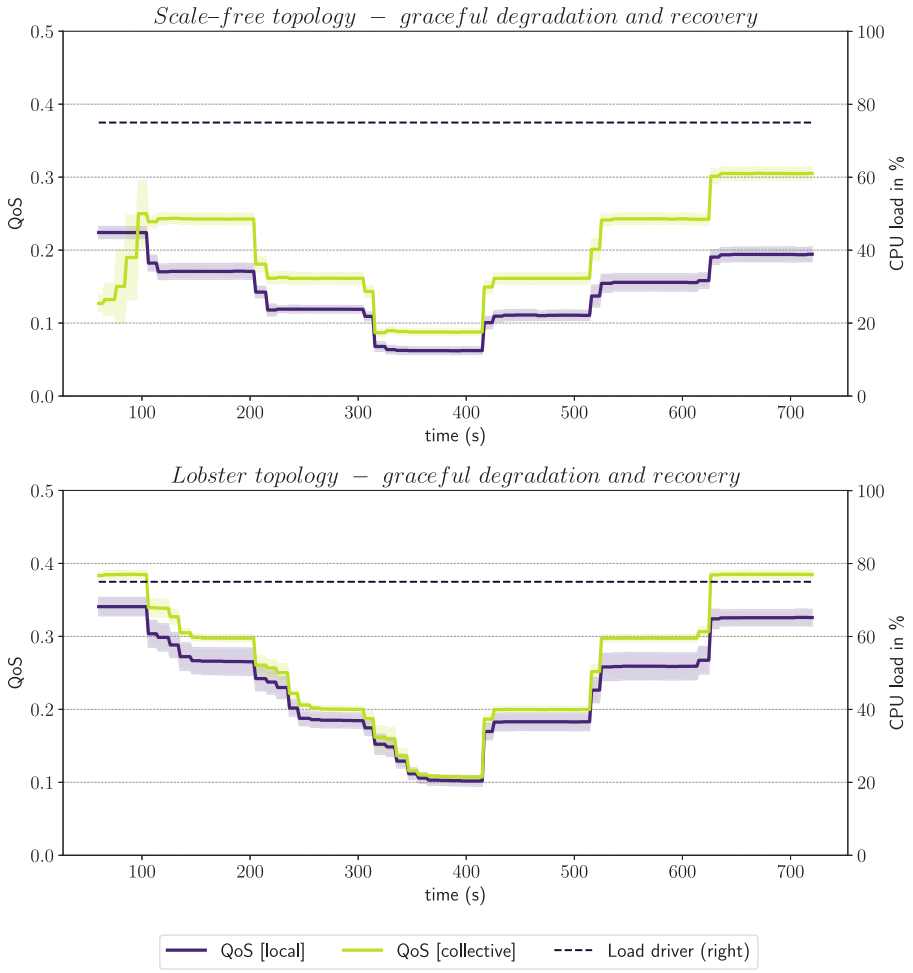

Fig. 7. Results for the graceful degradation and recovery scenario with a scale-free (upper) and lobster (lower) topology. The collective strategy is depicted in green, the local one in violet, and the load driver is dashed and black. Shadows around the lines represent $\pm 1\sigma$ of the value. The collective strategy shows better response to degradation and better recovery capabilities, particularly in denser networks where the probability of segmentation is lower.

always successful), symmetric latency, and that the latency is low enough compared to the rate at which the application operates. All these aspects call for a more in-depth and realistic evaluation, which is in our research roadmap.

7. Related work

This work contributes to distributed systems' reconfiguration by providing a middleware for pulverised systems and investigating global-level, decentralised reconfiguration heuristics expressed in aggregate computing. Therefore, the work lies at the intersection of different research threads: component models, application partitioning, application deployment and reconfiguration, and macro-programming languages. We stress that a comparison of pulverisation with different component models and partitioning solutions is beyond the scope of this paper; as a consequence, the comparison with other reconfiguration techniques (which are specific to component models and partitioning solutions) is also not relevant. Also, notice that we abstract from how pulverised systems are specified: applications might be designed as such up-front or automatically pulverised [12]. In the following, we provide an overview of related research efforts, to clarify the positioning of the contribution.

7.1. Component models and application partitioning

Regarding the specification of (non-monolithic) distributed systems, approaches typically adopt *component models* [5], or techniques that leverage them, such as *Architecture Description Languages (ADLs)* [39], *service-oriented frameworks* [40], and Statecharts [41] (or derived [42,43]) approaches. Relevant examples of component models include *Fractal* [44], which provides a hierarchical and reflective approach, *BIP* [45], which provides a flat and correct-by-construction approach, the *Gamma Statechart*

Composition Framework [46], which provides a composition language for the composition of statechart-based reactive components, and *STATEMATE* [47], which provides a graphical-oriented set of tools for the specification, analysis, design, and documentation of complex reactive systems. The pulverisation approach is not properly a component model but may *use* component models to specify a partitioning schema for distributed applications (cf. Section 3.2).

Application partitioning [6] is the activity of splitting an application into distinct components while retaining its original semantics; it could be a *manual* activity performed at the design phase, realised by reifying multiple executable and deployable components [5], or an *automatic* activity performed by software, offline or online [6]. For instance, in BIP, the application software has to be recast (e.g., by source translations) to a BIP model to be verified and operated at runtime. Partitioning approaches can be distinguished by granularity (e.g., components, classes, methods), partitioning model (e.g., graph-based, optimisation-based, hybrid), goal (e.g., improving performance, reducing footprint), time of the allocation decision (e.g., offline, online, hybrid), etc. [6]. Pulverisation works at the granularity of components, uses a graph-based model [11], can support multiple goals [12], and enables deployment decisions both offline and online—the latter possibility being shown in this paper.

7.2. Application deployment and reconfiguration

Regarding deployment and reconfiguration, proposals typically combine (i) one or more DSLs for specifying application architectures, deployments, and reconfiguration policies, and (ii) a middleware responsible for managing the system, often organised according to the *MAPE-K (Monitor-Analyse-Plan-Execute with Knowledge)* pattern [9]. Sometimes, approaches cover only some of these aspects. For instance, in the *multi-tier programming* language [48] like ScalaLoci [49] the developer explicitly specifies, in a single codebase, the different locations (*tiers)* and the deployment units for them, and the system statically checks the correctness of interaction; however, there is no support for dynamic reconfiguration. For instance, in BIP, a *deployment* is denoted as an assignment of components to a *map*, which is a set of locations upon which a connectivity relationship is defined.

An overview and survey of automatic deployment of distributed systems is provided by Arcangeli et al. in [50]. Approaches can be classified according to what kind of software is deployed (cf. component models and partitioning solutions), where, and how. A related, recent survey of formal methods and approaches for verification-oriented component-based reconfiguration is provided in [7]. As mentioned, reconfiguration approaches target different component models [7]. For the BIP component model, DR-BIP [45] has been proposed as an ADL for dynamically reconfigurable systems; it is based on the definition of *motifs*, which are parametrised around maps and encapsulate (i) behaviour, expressed as a set of components; (ii) interaction rules; and (iii) reconfiguration rules, expressed as guarded functions that change the contents of motifs. While pulverisation abstracts from how a deployment is designed and performed, a concrete implementation has to take these decisions: in this work, we propose a middleware that provides generic services for managing deployments, where the deployment logic can be expressed separately as aggregate programs.

Our approach to reconfiguration with aggregate computing is peculiar in that it targets pulverised systems (especially for large-scale CPSs and collective systems), and is based on self-organising reconfiguration policies that are decentralised, heuristic, and global. In the literature, the majority of approaches are *centralised*, which a single entity responsible for generating deployment plans. For instance, in *DELADAS* [51] and *ConfSolve* [52], *reconfiguration rules* are expressed in terms of *constraint sets* and deployment planning is recast to solving a constraint satisfaction problem by a centralised solver. The *ADSL (AWaRE DSL)* [53] is another work on constraint-based self-management, still based on a constraint solver and a central mapping orchestrator. A different approach, still based on centralised model checking, and not only focussed on deployment reconfiguration but generally on self-managing systems, is *ASSL* [54]: it provides a formal language to express *self-management policies* based on fluents (states), events, and actions; however, in case of large models, the authors recommended reducing the state graph for model checking to be viable.

Due to the emphasis on optimisation and constraint solving, few approaches address large-scale systems. One example is [55], a model-based approach for deployment of software on *fleets* of devices across the ECC. However, the work is not about the deployment of component-partitioned applications, but rather on the deployment of different versions of the same software. Also, it still adopts centralised constraint solving, showing limitations regarding scalability (e.g., taking several minutes to assign few deployment plans to few hundreds of devices). *MuScADeL* [56] shares similar goals to this work: supporting deployment of large and heterogeneous, geographically-dispersed, and multi-layer infrastructures. The constraints included in component specifications are used to produce *probe* artifacts that are exploited by the middleware to support data collection: higher-level probes query and aggregate data from lower-level probe. However, the generation of a deployment plan is still based on centralised constraint solving.

7.3. Macro-programming

Macro-programming is the umbrella term for programming approaches supporting developers in the specification of the macro-level/system-level/global behaviour of a network of interacting devices [17]. As reported in recent surveys [17,57], macro-programming approaches tend to address specific domains like Wireless Sensor Networks (WSNs), multi-robot systems, and the IoT. In the context of WSNs, much of the emphasis is on sensing, data routing, and data processing; moreover, often the behaviour is a *task graph*, and the deployment goal is to find a placement of tasks on a target network [26]. However, there are also a few general-purpose macro-programming approaches, like aggregate computing [13].

Characterising features of macro-programming solutions include the use of macro-level abstractions (e.g., collective data structures or spatial abstractions) and rules for setting up the link between micro-level and macro-level behaviour. With respect to macro-programming, the reconfiguration approaches covered in Section 7.2 differ from the micro-level descriptions (e.g., DR-BIP) or overly abstract specifications (e.g., constraint sets like in DELADAS and ConfSolve) typically addressed by a centralised orchestrator.

To the best of our knowledge, our work is the first one to propose a macro-programming solution for reconfiguration in the IoT domain, though specific to the pulverisation model.

Though other general-purpose macro-programming approaches exist, like *SCEL (Service Component Ensemble Language)* and derivates [58], aggregate computing has shown to be more practical and especially suitable for *self-organisation engineering* [18]. Indeed, thanks to its ability to capture macroscopic behaviour patterns as reusable functions of computational fields, it supports flexible specification of decentralised reconfiguration logic by a global perspective (cf. Listing 5). Additionally, formal results, such as the preservation of self-stabilisation across composition of self-stabilising aggregate program fragments [24], provide certain guarantees about the self-organising processes involved.

Regarding the effectiveness of aggregate programming for the reconfiguration task at hand, we may distinguish three aspects: productivity, scalability, and explainability. Productivity is enhanced by the macro-programming approach, the compositional nature of the paradigm, and the availability of practical DSLs that include comprehensive libraries of self-organising building blocks. Previous work provided preliminary comparative results showing benefits, in terms of reduced boilerplate code, with respect to actor-based and publish–subscribe solutions [59,60]. This is in contrast to rule-based approaches, which tend to adopt a "local" perspective that may not straightforwardly map to desired global outcomes, due to emergence. Scalability is promoted by the decentralised aggregate execution model (cf. Section 3.1) as well as patterns like SCR (cf. Section 5.2). Finally, explainability is inherent in the use of a language for expressing reconfiguration logic—which is not generally the case when machine learning-based techniques are adopted.

8. Conclusions and future work

The original pulverisation approach focuses on how to partition the system into smaller components, improving the separation between functional aspects and deployment concerns. Prior works on pulverisation [11,12] validated the partitioning and deployment approach on static deployments. In this paper, we focused on reconfiguration and accordingly proposed a middleware architecture for pulverised systems (cf. RQ1) and an aggregate computing-based approach for specifying reconfiguration policies (cf. RQ2, RQ3).

With respect to the research questions stated in Section 2, in this paper we:

1. answered the RQ1 by devising a general architecture structured in layers where each layer is responsible for a specific aspect of the system;
2. positively answered the RQ2 demonstrating the feasibility of Aggregate Computing to express decentralised reconfiguration policies from a global perspective. In particular, we have shown how the SCR pattern can be used to express reconfiguration policies in a succinct and high-level fashion. In this regard, we have conducted several experiments in different conditions, showing the effectiveness of the proposed approach in different network topologies and system conditions like device failure and variable load of the system;
3. show better performance results on average with the proposed approach compared to a local one, highlighting the benefits of Aggregate Computing to express reconfiguration policies, positively answering the RQ3.

With this work, we focused on the reconfiguration layer of the middleware architecture, making some assumptions about the other layers, like the one responsible for the network topology management and communication. A valuable next step would be to devise an implementation for the *infrastructure layer* investigating the adoption of distributed solutions for managing the network topology and communication. Then, provide a complete implementation of the middleware architecture where both the *application management layer* and the *infrastructure layer* cooperate to manage the system. A related research direction is the *detection* of anomalies that trigger reconfigurations and their *explanation* before reconfiguration. Detection could be performed in two ways: offline, through *static program analysis*, or online, through *monitoring*. In the former case, the specification is fed to a specialised tool that analyses the program and detects potential anomalies ahead of runtime execution; counteractions include a revision of the specification, or specially-crafted reconfiguration policies that tackle these statically-detectable anomalies before they manifest. The online case is arguably complimentary to the former, and its integration with the proposed architecture could in the future lead to an integrated framework which, besides responding to anomalies and self-stabilise to a new configuration, can provide hints to *prevent* anomalies altogether.

CRediT authorship contribution statement

Nicolas Farabegoli: Writing – review & editing, Writing – original draft, Validation, Software, Data curation, Conceptualization. **Danilo Pianini:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Conceptualization. **Roberto Casadei:** Writing – review & editing, Writing – original draft, Investigation, Conceptualization. **Mirko Viroli:** Writing – review & editing, Writing – original draft, Supervision, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Mirko Viroli reports financial support was provided by Italian Ministry of University and Research. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work has been supported by the Italian PRIN Project COMMON-WEARS (2020HCWWLP).

Data availability

No data was used for the research described in the article.

References

[1] D. Rosendo, A. Costan, P. Valduriez, G. Antoniu, Distributed intelligence on the edge-to-cloud continuum: A systematic literature review, J. Parallel Distrib. Comput. 166 (2022) 71–94, http://dx.doi.org/10.1016/J.JPDC.2022.04.004.
[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] 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.
[4] S. Dustdar, V. Casamayor-Pujol, P.K. Donta, On distributed computing continuum systems, IEEE Trans. Knowl. Data Eng. 35 (4) (2023) 4092–4105, http://dx.doi.org/10.1109/TKDE.2022.3142856.
[5] I. Crnkovic, S. Sentilles, A. Vulgarakis, M.R.V. Chaudron, A classification framework for software component models, IEEE Trans. Softw. Eng. 37 (5) (2011) 593–615, http://dx.doi.org/10.1109/TSE.2010.83.
[6] J. Liu, E. Ahmed, M. Shiraz, A. Gani, R. Buyya, A. Qureshi, Application partitioning algorithms in mobile cloud computing: Taxonomy, review and future directions, J. Netw. Comput. Appl. 48 (2015) 99–117, http://dx.doi.org/10.1016/j.jnca.2014.09.009.
[7] H. Coullon, L. Henrio, F. Loulergue, S. Robillard, Component-based distributed software reconfiguration: A verification-oriented survey, ACM Comput. Surv. (2023) http://dx.doi.org/10.1145/3595376.
[8] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: Yesterday, today, and tomorrow, in: Present and Ulterior Software Engineering, Springer, 2017, pp. 195–216, http://dx.doi.org/10.1007/978-3-319-67425-4_12.
[9] J.O. Kephart, D.M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50, http://dx.doi.org/10.1109/MC.2003.1160055.
[10] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Trans. Auton. Adapt. Syst. 4 (2) (2009) 14:1–14:42, http://dx.doi.org/10.1145/1516533.1516538.
[11] R. Casadei, D. Pianini, A. Placuzzi, M. Viroli, D. Weyns, Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment, Future Internet 12 (11) (2020) 203, http://dx.doi.org/10.3390/FI12110203.
[12] R. Casadei, G. Fortino, D. Pianini, A. Placuzzi, C. Savaglio, M. Viroli, A methodology and simulation-based toolchain for estimating deployment performance of smart collective services at the edge, IEEE Internet Things J. 9 (20) (2022) 20136–20148, http://dx.doi.org/10.1109/JIOT.2022.3172470.
[13] J. Beal, D. Pianini, M. Viroli, Aggregate programming for the internet of things, Computer 48 (9) (2015) 22–30, http://dx.doi.org/10.1109/MC.2015.261.
[14] N. Farabegoli, Nicolasfara/reconfiguration-experiments: 1.0.0, 2023, http://dx.doi.org/10.5281/zenodo.10372019.
[15] G.R. Russo, V. Cardellini, F.L. Presti, Reinforcement learning based policies for elastic stream processing on heterogeneous resources, in: 13th ACM International Conference on Distributed and Event-Based Systems, DEBS 2019, Proceedings, ACM, 2019, pp. 31–42, http://dx.doi.org/10.1145/3328905.3329506.
[16] K. Zhang, Z. Yang, T. Başar, Multi-agent reinforcement learning: A selective overview of theories and algorithms, in: Studies in Systems, Decision and Control, Springer International Publishing, 2021, pp. 321–384, http://dx.doi.org/10.1007/978-3-030-60990-0_12.
[17] 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.
[18] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, D. Pianini, From distributed coordination to field calculus and aggregate computing, J. Log. Algebr. Methods Program. 109 (2019) 100486, http://dx.doi.org/10.1016/j.jlamp.2019.100486.
[19] 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.
[20] D. Pianini, S. Montagna, M. Viroli, Chemical-oriented simulation of computational systems with ALCHEMIST, J. Simul. 7 (3) (2013) 202–215, http://dx.doi.org/10.1057/jos.2012.27.
[21] G. Audrito, F. Terraneo, W. Fornaciari, FCPP+Miosix: Scaling aggregate programming to embedded systems, IEEE Trans. Parallel Distrib. Syst. 34 (3) (2023) 869–880, http://dx.doi.org/10.1109/TPDS.2022.3232633.
[22] L. Testa, G. Audrito, F. Damiani, G. Torta, Aggregate processes as distributed adaptive services for the industrial internet of things, Perv. Mob. Comput. 85 (2022) 101658, http://dx.doi.org/10.1016/J.PMCJ.2022.101658.
[23] M. Mamei, F. Zambonelli, L. Leonardi, Co-fields: A physically inspired approach to motion coordination, IEEE Perv. Comput. 3 (2) (2004) 52–61, http://dx.doi.org/10.1109/MPRV.2004.1316820.
[24] 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.
[25] K. Tumer, D.H. Wolpert, A survey of collectives, in: K. Tumer, D.H. Wolpert (Eds.), Collectives and the Design of Complex Systems, Springer Science & Business Media, 2004, pp. 1–42, http://dx.doi.org/10.1007/978-1-4419-8909-3_1.
[26] A. Pathak, V.K. Prasanna, Energy-efficient task mapping for data-driven sensor network macroprogramming, IEEE Trans. Comput. 59 (7) (2010) 955–968, http://dx.doi.org/10.1109/TC.2009.168.
[27] E. Bainomugisha, A.L. Carreton, T.V. Cutsem, S. Mostinckx, W.D. Meuter, A survey on reactive programming, ACM Comput. Surv. 45 (4) (2013) 52:1–52:34, http://dx.doi.org/10.1145/2501654.2501666.
[28] 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.
[29] F. Aït-Salaht, F. Desprez, A. Lebre, An overview of service placement problem in fog and edge computing, ACM Comput. Surv. 53 (3) (2021) 65:1–65:35, http://dx.doi.org/10.1145/3391196.
[30] G. Audrito, M. Viroli, F. Damiani, D. Pianini, J. Beal, A higher-order calculus of computational fields, ACM Trans. Comput. Log. 20 (1) (2019) 5:1–5:55, http://dx.doi.org/10.1145/3285956.
[31] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644, http://dx.doi.org/10.1145/361179.361202.
[32] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, D. Pianini, From field-based coordination to aggregate computing, in: Coordination Models and Languages - 20th IFIP WG 6.1 International Conference, COORDINATION 2018, Proceedings, in: LNCS, vol. 10852, Springer, 2018, pp. 252–279, http://dx.doi.org/10.1007/978-3-319-92408-3_12.

[33] D. Pianini, R. Casadei, M. Viroli, A. Natali, Partitioned integration and coordination via the self-organising coordination regions pattern, Future Gener. Comput. Syst. 114 (2021) 44–68, http://dx.doi.org/10.1016/J.FUTURE.2020.07.032.
[34] 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, Proceedings, IEEE, 2022, pp. 81–90, http://dx.doi.org/10.1109/ACSOS55765.2022.00026.
[35] A.-L. Barabási, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512, http://dx.doi.org/10.1126/science.286.5439.509.
[36] X. Zhou, B. Yao, X. Chen, Every lobster is odd-elegant, Inform. Process. Lett. 113 (1–2) (2013) 30–33, http://dx.doi.org/10.1016/J.IPL.2012.09.008.
[37] S. Hoyer, J. Hamman, xarray: N-D labeled arrays and datasets in Python, J. Open Res. Softw. 5 (1) (2017) http://dx.doi.org/10.5334/jors.148.
[38] J.D. Hunter, Matplotlib: A 2D graphics environment, Comput. Sci. Eng. 9 (3) (2007) 90–95, http://dx.doi.org/10.1109/MCSE.2007.55.
[39] N. Medvidovic, R.N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Trans. Softw. Eng. 26 (1) (2000) 70–93, http://dx.doi.org/10.1109/32.825767.
[40] A.L. Lemos, F. Daniel, B. Benatallah, Web service composition: A survey of techniques and tools, ACM Comput. Surv. 48 (3) (2016) 33:1–33:41, http://dx.doi.org/10.1145/2831270.
[41] D. Harel, Statecharts: A visual formalism for complex systems, Sci. Comput. Program. 8 (3) (1987) 231–274, http://dx.doi.org/10.1016/0167-6423(87)90035-9.
[42] S.D. Dewasurendra, Statecharts for reconfigurable control of complex reactive systems: a new formal verification methodology, in: First International Conference on Industrial and Information Systems, IEEE, 2006, pp. 274–278, http://dx.doi.org/10.1109/iciis.2006.365736.
[43] P. Sánchez, B. Álvarez, R. Martínez, A. Iborra, Embedding statecharts into Teleo-reactive programs to model interactions between agents, J. Syst. Softw. 131 (2017) 78–97, http://dx.doi.org/10.1016/J.JSS.2017.05.081.
[44] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The FRACTAL component model and its support in Java, Softw. - Pract. Exp. 36 (11–12) (2006) 1257–1284, http://dx.doi.org/10.1002/spe.767.
[45] R.E. Ballouli, S. Bensalem, M. Bozga, J. Sifakis, Four exercises in programming dynamic reconfigurable systems: Methodology and solution in DR-BIP, in: Leveraging Applications of Formal Methods, Verification and Validation. ISoLA, Proceedings, Part III, in: Lecture Notes in Computer Science, vol. 11246, Springer, 2018, pp. 304–320, http://dx.doi.org/10.1007/978-3-030-03424-5_20.
[46] B. Graics, V. Molnár, A. Vörös, I. Majzik, D. Varró, Mixed-semantics composition of statecharts for the component-based design of reactive systems, Softw. Syst. Model. 19 (6) (2020) 1483–1517, http://dx.doi.org/10.1007/S10270-020-00806-5.
[47] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M.B. Trakhtenbrot, STATEMATE: a working environment for the development of complex reactive systems, IEEE Trans. Softw. Eng. 16 (4) (1990) 403–414, http://dx.doi.org/10.1109/32.54292.
[48] P. Weisenburger, J. Wirth, G. Salvaneschi, A survey of multitier programming, ACM Comput. Surv. 53 (4) (2021) 81:1–81:35, http://dx.doi.org/10.1145/3397495.
[49] P. Weisenburger, M. Köhler, G. Salvaneschi, Distributed system development with ScalaLoci, Proc. ACM Program. Lang. 2 (OOPSLA) (2018) 129:1–129:30, http://dx.doi.org/10.1145/3276499.
[50] J. Arcangeli, R. Boujbel, S. Leriche, Automatic deployment of distributed software systems: Definitions and state of the art, J. Syst. Softw. 103 (2015) 198–218, http://dx.doi.org/10.1016/j.jss.2015.01.040.
[51] A. Dearle, G. Kirby, A. McCarthy, A framework for constraint-based deployment and autonomic management of distributed applications, in: 1st International Conference on Autonomic Computing (ICAC 2004). Proceedings, IEEE, 2004, pp. 300–301, http://dx.doi.org/10.1109/ICAC.2004.3.
[52] J.A. Hewson, P. Anderson, A.D. Gordon, Constraint-based autonomic reconfiguration, in: SASO, IEEE Computer Society, 2013, pp. 101–110, http://dx.doi.org/10.1109/SASO.2013.23.
[53] M.B. Chhetri, H.P. Luong, A.V. Uzunov, Q.B. Vo, R. Kowalczyk, S. Nepal, I. Rajapakse, ADSL: an embedded domain-specific language for constraint-based distributed self-management, in: ASWEC, IEEE Computer Society, 2018, pp. 101–110, http://dx.doi.org/10.1109/ASWEC.2018.00022.
[54] E. Vassev, M. Hinchey, The ASSL approach to formal specification of self-managing systems, in: Models, Mindsets, Meta: The What, the how, and the Why Not?, in: Lecture Notes in Computer Science, vol. 11200, Springer, 2018, pp. 268–296, http://dx.doi.org/10.1007/978-3-030-22348-9_17.
[55] H. Song, R. Dautov, N. Ferry, A. Solberg, F. Fleurey, Model-based fleet deployment in the IoT-edge-cloud continuum, Softw. Syst. Model. 21 (5) (2022) 1931–1956, http://dx.doi.org/10.1007/s10270-022-01006-z.
[56] R. Boujbel, S. Leriche, J.-P. Arcangeli, A framework for autonomic software deployment of multiscale systems, Int. J. Adv. Softw. 7 (1 & 2) (2014) 353–369, URL https://hal.science/hal-03256404.
[57] I.G.S. Júnior, T.S.d. Santana, R.d.F. Bulcão-Neto, B.F. Porter, The state of the art of macroprogramming in IoT: An update, J. Internet Serv. Appl. 13 (1) (2022) 54–65, http://dx.doi.org/10.5753/jisa.2022.2372.
[58] 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.
[59] G. Audrito, R. Casadei, F. Damiani, G. Salvaneschi, M. Viroli, Functional programming for distributed systems with XC, in: K. Ali, J. Vitek (Eds.), 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.
[60] R. Casadei, F. Damiani, G. Torta, M. Viroli, Actor-based designs for distributed self-organisation programming, in: Active Object Languages: Current Research Trends, in: Lecture Notes in Computer Science, vol. 14360, Springer, 2024, pp. 37–58, http://dx.doi.org/10.1007/978-3-031-51060-1_2.