future internet

MDPI

Article

Pulverization in Cyber-Physical Systems: Engineering the Self-Organizing Logic Separated from Deployment

Roberto Casadei [1], Danilo Pianini [1,*], Andrea Placuzzi [1], Mirko Viroli [1] and Danny Weyns [2,3]

[1] Department of Computer Science and Engineering (DISI), Alma Mater Studiorum–Università di Bologna, 47521 Cesena FC, Italy; roby.casadei@unibo.it (R.C.); andrea.placuzzi@unibo.it (A.P.); mirko.viroli@unibo.it (M.V.)
[2] Department of Computer Science, Katholieke Universiteit Leuven, 3000 Leuven, Belgium; danny.weyns@kuleuven.be
[3] Department of Computer Science, Linnaeus University, 351 95 Växjö, Sweden
* Correspondence: danilo.pianini@unibo.it

Received: 1 October 2020; Accepted: 16 November 2020; Published: 19 November 2020

Abstract: Emerging cyber-physical systems, such as robot swarms, crowds of augmented people, and smart cities, require well-crafted self-organizing behavior to properly deal with dynamic environments and pervasive disturbances. However, the infrastructures providing networking and computing services to support these systems are becoming increasingly complex, layered and heterogeneous—consider the case of the edge–fog–cloud interplay. This typically hinders the application of self-organizing mechanisms and patterns, which are often designed to work on flat networks. To promote reuse of behavior and flexibility in infrastructure exploitation, we argue that self-organizing logic should be largely *independent* of the specific application deployment. We show that this separation of concerns can be achieved through a proposed "*pulverization approach*": the global system behavior of application services gets broken into smaller computational pieces that are continuously executed across the available hosts. This model can then be instantiated in the aggregate computing framework, whereby self-organizing behavior is specified compositionally. We showcase how the proposed approach enables expressing the application logic of a self-organizing cyber-physical system in a deployment-independent fashion, and simulate its deployment on multiple heterogeneous infrastructures that include cloud, edge, and LoRaWAN network elements.

Keywords: self-organization; decentralized control; deployment independence; declarative programming; aggregate computing

1. Introduction

Among the many approaches proposed to engineer systems featuring distributed intelligence, a relevant one is *self-organization* [1], by which global structure and behavior are robustly achieved by continuous local interaction of simple individual components. This is generally meant to promote inherent adaptation to unexpected (or not completely foreseeable) contingencies, supporting applications in contexts such as human social behavior, swarm robotics, and task allocation. Therefore, in this paper we deal with artificial (software-based) self-organizing systems. Self-organization was first observed in nature, in natural systems such as ant colonies or bird flocks, and the first mechanisms for self-organization engineering were largely inspired by such systems [1]. In recent decades, the engineering of self-organization has been widely researched in various areas including multi-agent systems [2], autonomic computing [3], self-managing and adaptive systems [4],

and resource management [5]. Self-organization is generally understood as the ability of a system to sustain its internal order, in face of change and without any external control—few formalization attempts exist, e.g., via a notion of entropy of the population of system components [6]. Often, it is common in the literature to refer to artificial systems endowing the ability to self-regulate their internal structures and behavior (e.g., by mimicking self-organization mechanisms observed in nature) as simply *self-organizing* [3,6–9]. Following this convention, in this paper we use term "self-organizing (cyber-physical) system" to mean an "artificial (cyber-physical) system featuring self-organizing logic".

When self-organizing approaches are applied to distributed cyber-physical systems (CPS), they are essentially used to collect and process information generated by sensors distributed over the environment, and use it to control the system behavior itself [10]—often in a purely decentralized fashion. There, individual system components form a flat space that interact with each other based on physical proximity. In these settings, direct communications with centralized servers are unpractical, expensive, or even impossible.

Recent advances in technology foster pervasiveness of computing and communication, providing opportunities for novel services and applications, which then turn into new engineering and research challenges. Current trends are making modern CPS increasingly large-scale, heterogeneous, and dynamic, requiring increasingly complex and heterogeneous infrastructures [11]. On the one hand, remote clouds provide virtually unlimited on-demand resources (computing, storage, services), provided that costs (both in money and latency) and data protection procedures can meet the requirements of the application at hand. On the other hand, edge computing brings resources closer to users, reducing latency (increasing reactivity) and managing concerns about data dissemination. These two complementary architectures are often intertwined with other communication technologies, such as proximity interaction (e.g., Bluetooth or Near-field communication), and long-range wireless communication (e.g., Sigfox [12] or LoRaWAN [13]).

Such intricate and dynamic infrastructures complicate the task of solid engineering of distributed intelligent systems, whose logic tends to be significantly dependent on it. In particular, this hampers the possibility of reusing design elements across different application scenarios, and exploiting the available ICT resources opportunistically (e.g., for optimization, graceful degradation, or customization purposes). To that end, in this paper we investigate the following problem:

How to design and implement the self-organization logic of CPS in a deployment-independent way?

With "deployment-independent" we mean a separation of concerns between the self-organization logic and the deployment context. In particular, the behavioral description of the self-organization logic remains unchanged regardless of the specifics of the deployment context.

Consider as a motivating example, the problem of a smart-city CPS that consists of a computing ecosystem with devices that monitor and aggregate pollution levels and household temperatures to promote eco-friendly living. One typical approach to implement the application logic of such systems is through self-organizing algorithms—see for instance [14]. Such algorithms employ the spatiotemporal proximity of devices to regulate the flow of information in the system in a scalable way, ultimately promoting the emergence of smart collective behavior. Evidently, such applications can be set up using different heterogeneous sets of computing and networking devices. Depending on the concrete technology at hand, the communication between physically close devices may need to be implemented in different ways, for instance, based on direct device-to-device interaction or communication via intermediaries such as fog servers or the cloud. Ideally, the application logic should not be affected by the heterogeneity of the underlying technology or the specific deployment choices (cf., cost, performance, and reliability considerations). We use this example as a case in Section 6.

To tackle this type of problems, we propose a framework to design distributed adaptive behavior for large-scale CPSs rooted on what we call the *pulverization approach*. This approach breaks the overall system behavior into tiny pieces of computation logically linked to sensors, actuators, and neighboring components, each continuously scheduled and executed across space

(i.e., the computational devices available) and time.This "network" can then be smoothly mapped onto a variety of multi-layered deployment infrastructures. In other words, we provide a schema for engineering the self-organizing logic in CPSs: it is based on a flexible logical model which can be decomposed into a set of sub-components with well-defined relationships that can be deployed and wired separately (i.e., they work as "logical units of deployment"). Such a schema is to be properly instantiated and filled with details to actually drive self-organization. To that end, we show how this approach can be implemented in the framework of Aggregate Computing [15], where global self-organizing behavior can be specified declaratively, i.e., by composing pure functions expressing increasingly complex distributed algorithms. As a key benefit, the approach can make the Aggregate Computing toolchain [16,17], distributed algorithms and systems [18,19], readily available to work in the larger class of complex ICT infrastructures [20].

To showcase the pulverization proposal, we evaluate the approach by simulating a situated CPS, whose software components are deployed using a synergy of technologies including edge servers, low-power/long-range communication via LoRaWAN [13], MQTT (Message Queuing Telemetry Transport) [21,22], and cloud offloading. We run the system over different deployment configurations, exploiting the edge and cloud layers. Then, we verify that its self-organizing logic, written once and orthogonal to the specifics of deployment, always produce the same functional behavior—although, non-functional aspects such as performance (e.g., network response time) or operating cost are affected by the underlying deployment at hand. The evaluation is performed on an evolved version of the DingNet exemplar [23], augmented with Aggregate Computing support and features to select deployment schemes.

After discussing background and related work (Section 2), this paper describes the following key contributions:

1. A *novel model* for self-organizing cyber-physical systems that fosters "pulverized" execution to realize deployment independence, formalized by a structural operational semantics defining a transition system for logical and physical network of devices (Section 3);
2. An *instantiation of the model* for the aggregate computing framework (Section 4);
3. An *evaluation* of the pulverization approach (Section 6) on a simulated smart-city case comprising multiple deployments of a CPS for pollution-aware household heating control.

2. Background and Related Work

This section is structured in three parts that introduce the main related efforts and positions our work on this landscape: self-organization, decentralized self-adaptation, and flexible deployments of adaptive systems.

2.1. Self-Organization

A self-organizing system consists of (simple) components that realize the system goals through repeated local interaction. Self-organization [24] often draws inspiration from natural systems and promotes *emergence* [25,26], i.e., the creation of novel macro-level effects out of decentralized micro-level activity. Swarm intelligence exhibited by ants is probably the best-known example [27]. Such macro-effects are often connected with the ability of the system to be resilient to environmental perturbations (ranging from human in-the-loop effects to faults in physical devices or communications—which are the norm in large-scale distributed systems), or to deal with changing functional/non-functional requirements that have to be satisfied in a wide variety of conditions. Relevant application of self-organization include: study of human social behavior [28], crowd tracking and steering [15], energy demand allocation [29], terrain exploration [30], smart camera coverage [31], task allocation of autonomous vehicles [32], and ICT resources coordination [33].

Most of these works show a trend in self-organization, shifting from natural inspiration to the identification of "artificial" mechanisms for engineering the collective part of adaptive systems,

namely to develop so-called *collective adaptive systems* (CASs). Research on CASs [34] study how a *collective* (i.e., a dynamic group of "similar" autonomous entities—e.g., a swarm of robots) can support macro-level adaptivity and how such insights can be leveraged for engineering large-scale socio-technical systems.

The approach studied in this paper takes somewhat inspiration from these latter approaches, looking for ways to abstract from the actual deployment of physical systems, hence aiming to apply self-organization techniques in heterogeneous deployment settings such as those relying on multiple layers of cloud/fog/edge computing.

2.2. Decentralized Self-Adaptation

A *self-adaptive system* is one that is equipped with an external feedback loop that tracks the state of the system and its environment and, through adapting it to changes, ensures that a set of goals is achieved, or if necessary, the system is gracefully degraded [35–39]. In this paper, we are concerned with self-adaptive *software* systems [40] as well as their deployment, in particular cyber-physical systems.

Research on self-adaptive software relates to the field of *autonomic computing* [41]. A self-adaptive (or autonomic) system can be thought as consisting of three main pieces [38]: (i) a *managed system*, which provides the application logic and should be adapted to satisfy its requirements in different circumstances; (ii) an operating *environment*, which is generally characterized by *uncertainty* in terms of changes that are difficult to predict; and (iii) a *managing system*, (also known as *autonomic manager* [41] or *adaptation engine* [40]), that is responsible for the management of the managed system. A common approach to realize self-adaptation is architecture-based adaptation [37,42,43], which leverages architectural models of the system at runtime [44]. The managing system is usually organized around in a feedback loop that realizes four main functions [41]: *Monitoring*, *Analysis*, *Planning*, and *Execution*, that share common *Knowledge*, MAPE-K in short.

The work presented in this paper is in particular related to *decentralized self-adaptive systems*, i.e., systems in which there is not a single component responsible for adaptation, but adaptation control is distributed among multiple components [45]. In contrast to a self-organizing system, a decentralized self-adaptive system consists of components that are equipped with feedback loops that interact locally to adapt the underlying components to realize the system goals.

Decentralization in control fosters *scalability* (avoiding bottlenecks), *robustness* (avoiding single points of failure), and *efficiency through the exploitation of locality* (avoiding the overhead of non-local interactions). Sometimes, centralized control is simply not an option for a large class of modern distributed systems, because of cost of impact on quality attributes [4,46].

Despite the high relevance of decentralization of control in self-adaptive systems, existing work remains limited. We highlight a few representative approaches. Malek et al. [47] use an auction-based coordination mechanism to find the appropriate deployment architecture under changing operating conditions. Vromant et al. [48] extends MAPE loops with support for inter-loop and intra-loop coordination. Vogel and Giese [49] present *EUREMA (ExecUtable RuntimE MegAmodels)* that offers support for specifying interacting feedback loops models that then can be directly deployed and interpreted by an interpreter. A related, but formally founded approach is presented in [50]. Calinescu et al. [51] present *DECIDE (DEcentralized Control In Distributed sElf-adaptive software)*, an approach to decentralize feedback loops that uses quantitative verification at runtime to assure Quality-of-Service (QoS) requirements in the presence of change. The approach demonstrates better scalability compared to a centralized approach, but only for small component models.

Weyns and Georgeff [52] apply self-adaptation in a multi-agent system application for automated guided vehicles. An example of self-adaptive middleware architecture for dynamic service composition is described in [53], and *GoPrime* [54] offers a decentralized middleware for self-assembly of distributed services. A gossip protocol realizes decentralized data dissemination to maintain an assembly of services that fulfils global QoS goals.

Although these approaches offer complementary contributions to tackle the complex problem of decentralization of control in self-adaptation, none of these approaches targets large-scale adaptive systems nor their deployment independence, which is exactly the contribution of this paper.

2.3. Flexible Deployment

In *distributed* systems, software components are deployed to hardware elements, and this affects the notion of "locality", and the connection of computations to physical space, in ways that deserve attention. We generally refer to the hardware and the media supporting communication between hardware elements (in general, these hardware element are part of the communication network and the physical environment, where principles of stigmergy are applied [55] to support the exchange of information of sensors and actuators) as *(ICT) infrastructure*. The problem of *deployment* is about how to map (components of) the system to available infrastructure. We use the same terminology of major literature [4,56], where term "distribution" refers to deployment and term "decentralization" refers to the degree by which *adaptation control* is handled by multiple interacting components rather than by a single, central component. Notice that decentralization of control is *orthogonal* to distribution (i.e., to deployment) [4,56]: this is indeed the main theme of this article, where we show how this principle can actually be implemented, methodologically. Though deployment is considered a research challenge for self-adaptive systems [56], few works address the tension between application logic, adaptation control, and deployment.

Presently, infrastructure is increasingly heterogeneous, complex, and dynamic: computers range from low-powered, thin devices to large supercomputers; telecommunications technology can be wired or wireless, with many subtypes providing a wide range of capabilities and guarantees; storage and computational resources may be close, distant, or very distant to a communication endpoint—cf. edge, fog, and cloud computing; and finally, failures, interferences, replacements, extensions, can make the infrastructure dynamic. This richness is an opportunity but also a challenge for distributed computing paradigms, as typically each of them tends to address a specific deployment style.

At a first instance, applications should aim at being agnostic with respect to the underlying infrastructure and deployment. This is supported by *abstraction*, hiding unnecessary details for better focusing on the problem at hand, *and* by a *middleware* that makes the abstraction operational—filling the "gap" with the underlying platform or infrastructure where the details do matter.

This paper addresses the problem of identifying a suitable abstraction level to support deployment independence, or deployment agnosticism, in self-organizing systems. Instead of trying to match specific self-organizing models or patterns to the deployment at hand, we aim at a model by which general self-organizing strategies can be mapped to a variety of contexts. The key idea, inherited by approaches of so-called macro-programming [15,57,58], is to consider system behavior as a conceptually global computation performed on the whole set of system devices, and being effectively independent on details such as number and (mutual) location of devices. Declarative programming approaches can naturally be applied to promote the specification of *what* the computational system should perform, abstracting from concrete aspects pertaining lower-level details. Facilitated by declarativity, one such macro-behavior should then be "pulverized" into smaller pieces or chunks of behavior, each to be executed in a continuous way across "space", i.e., across all the devices of a system. According to the self-organization principle, such chunks affect only a logical locality, in terms of perception/action over the local portion of the environment, and exchange of messages with neighbors. Deployment independence is then achieved by a suitable mapping of a general self-organization problem to the actual platform, and performed by spreading such chunks in the available devices.

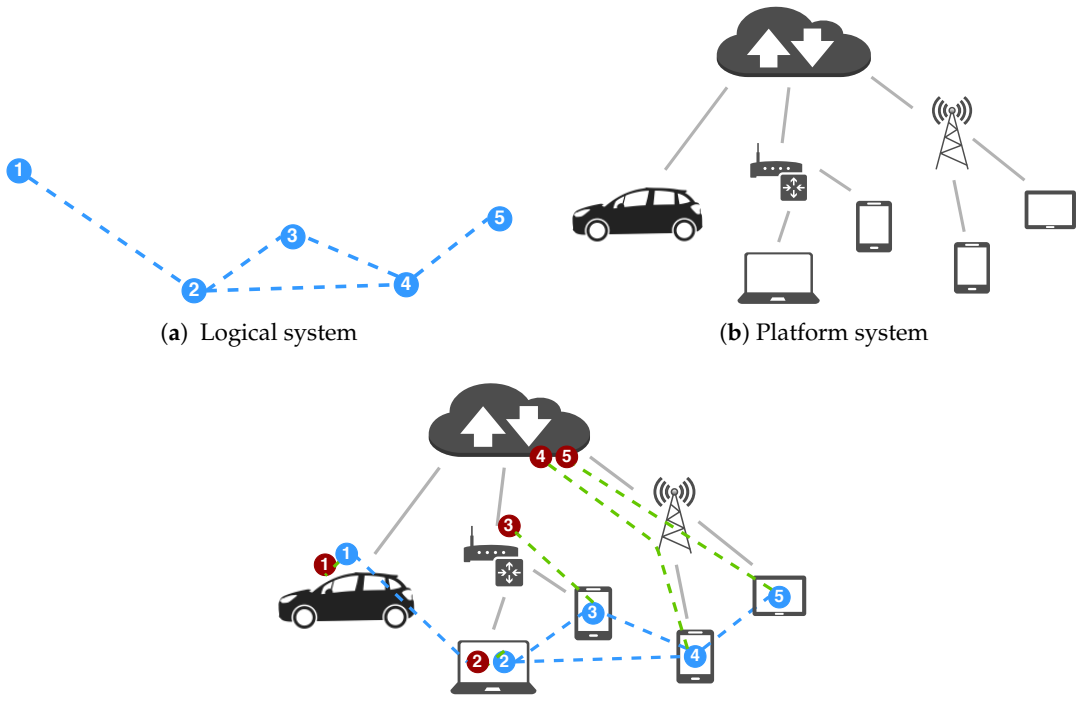
Large-scale distributed and self-adaptive systems as envisioned in forthcoming CPS and the Internet-of-Things (IoT) make a case for *distributed declarative programming* promoting self-organization, while at the same time leaving *degrees of freedom* [59] to designers and deployment. We highlight several representative research efforts on deployment and automatic reconfiguration of systems. *Osmotic computing* [11] is an approach to opportunistic deployment of microservices on the

edge-fog-cloud platform. An osmotic platform aims to reach and maintain an "osmotic equilibrium" between infrastructural and application requirements by automatically migrating microservices to deployment locations. However, the approach mainly targets centrally orchestrated systems. Other approaches leverage component-based, architectural descriptions to decouple application logic and deployment. For instance, *DR-BIP (Dynamically Reconfigurable Behavior Interaction Priority model)* [60] and *DReAM (Dynamically Reconfigurable Architectural Modelling)* [61] use *components* (capturing behavior), *connectors* (capturing interaction between components' *ports*), *maps* (logical topologies), and *deployments* (associating components to map locations), overall organized in *motifs* (dynamic architectural configurations), to model and analyze dynamic architectures.

These approaches have some similarity with the approach presented in this paper, but they are arguably more complex and our approach explicitly addresses self-organizing CPS.

3. A Model of Deployment-Independent, Self-Organizing, Cyber-Physical Systems

In this paper, we present an approach to conceive self-organization in distributed systems that facilitates deployment independence, i.e., the ability of an application to run with no change on various deployments while retaining its original functional semantics. It is rooted in the idea (illustrated in Figure 1) to organize the structure and behavior of a system so that (i) the developer can focus on a logical model that mostly abstracts details concerning deployment, scheduling and communication; and (ii) the logical model can be straightforwardly partitioned into a set of deployable software components, which deployers (e.g., devops team members or automated systems) can "freely" distribute on available infrastructure. The application logic will obtain the functional goals independent of the actual deployment, yet the choice of the deployment at hand typically affects non-functional properties, such as performance and cost.



(**a**) Logical system (**b**) Platform system

(**c**) Pulverization and binding

Figure 1. Deployment independence via pulverization. Designers of the business logic target the logical system (**a**), whose abstractions are identifiable nodes and neighbors. Once the deployment target (composed of hosts and communication channels) is identified (**b**), logical devices are *pulverized* into sub-components, which are then mapped to hosts (**c**). In the latter figure, green dashed lines represent network interactions between single-device sub-components.

Let us consider a cyber-physical system consisting of: a *cyber system* (Section 3.1), which provides a logical model for the system; and a *platform system*, which models networks of hosts for deployment (hence subsuming physical systems as well as simulation platforms) and to which the cyber system can be mapped to by a so-called *deployment* (Section 3.2). In the following, we describe this model informally at first and then formally by a structural operational semantics [62], e.g., in the style of calculi like π-calculus [63] (the quintessential process algebra for mobility) or the field calculus [64] (the main foundation of aggregate computing). The main goal of the formalization, presented incrementally in the following, is to provide an unambiguous specification of what constitutes pulverization, clarify subtle aspects of the model, and state the deployment-independence property rigorously. The model will then be instantiated in the macro-programming approach of aggregate computing in Section 4.

3.1. Cyber System

A cyber system is a collection of *logical devices* (or devices for short), each with the ability of connecting to other devices, called *neighbors*—generally, such connections (*logical neighboring links*) define a structure that can change over time. A device (Figure 2) is a logical entity with the following components:

- A set σ of logical *sensors*.
- A set α of logical *actuators*.
- A *state* κ, representing the device's local knowledge (where we abstract the particular representation chosen for a state).
- A *communication component* χ handling interaction with neighbors, holding information on the identity of neighbors and how to reach them, managing *input channels* used to receive external messages into the device's state, and *output channels* for emitting messages to all its neighbors (i.e., the output channel of a device is connected to the input channels of all its neighbor devices as per the neighboring relationship).
- A *computation* function β modelling the device *behavior*, which maps the state of the device to (i) a new state, (ii) a "prescriptive" set of actuations to be performed, and (iii) coordination messages to be emitted.

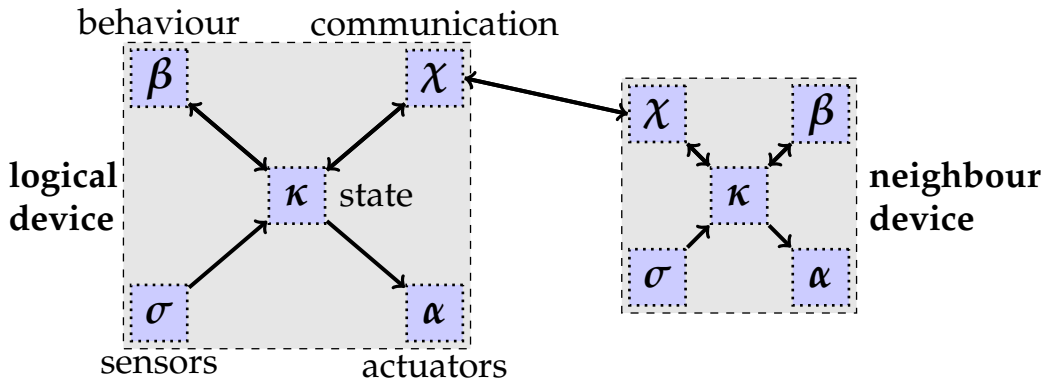


Figure 2. A logical device, split into sub-components, and one of its neighbors.

Each individual device of the cyber system performs a MAPE-like cycle that includes the following steps and that defines the interactions between the device's subcomponents as depicted in Figure 2 (with arrows denoting message flow).

1. *Context acquisition.* In this step, the device retrieves information from context sources (sensors and communications) and stores them in the device state.
2. *Computation.* The behavior function is applied against (data inferred from) the device state; its (possibly processed) output is stored in the device state.

3. *Coordination data propagation.* From the device state, coordination data is sent to all neighbors.
4. *Actuation.* Actuators are activated to execute a set of actions inferred from the device state (these actions are, e.g., prescribed through the device behavior or mere sensor data-driven reactions).

3.1.1. Formalization: Device Semantics

We formalize the main structure and behavioral elements of a cyber system by an operational semantics describing how the various components of a logical device interact with each other (and then how the whole logical network correspondingly evolves—Section 3.1.2). In doing so, we specifically identify how inner components are abstracted, hence clarifying their boundary, role and interactions.

We adopt several mathematical conventions frequently used in works in process algebras [63] and core calculi for programming languages and concurrent systems [64], which we here briefly recap. To capture the behavior of interactive systems we use labelled transition systems [63] which are triples $\langle State, \rightarrow, Label \rangle$ of a set $State$ of states, a ternary relation $\rightarrow \subseteq State \times Label \times State$, and a set $Label$ of labels. We write $st \xrightarrow{\ell} st'$ as a shorthand for $(st, \ell, st') \in \rightarrow$, meaning that the modelled system moves from state st to st' by an observable action ℓ. To describe interactive behavior we then give axioms and rules defining the triples in the relation $\rightarrow$.

We introduce meta-variables for the various element types of the model, which are used both as non-terminal symbols in grammars and in rules of the operational semantics: we let meta-variable s range over values produced by sensors, a over values sent to actuators, x over states, e over coordination messages (also called *exports*) providing an internal representation of the set of messages to send/receive, i over logical device identifiers, and v over message values exchanged with other devices. We abstract away the syntax of the elements that such meta-variables range over; on the other hand, when other elements need an abstract syntax, we introduce their meta-variables by non-terminal symbols of grammars instead. As additional notation, for any meta-variable, say v, we let decorations (v', v'', v_a, and so on) range over the same elements as v, let $Set(v)$ be the set of elements it ranges over, $v_\perp \in Set(v)$ be used as default value for $Set(v)$—unless differently specified, it is assumed that different meta-variables have disjoint sets of elements they range over.

To describe the components of a cyber system we introduce the following abstract grammar:

$$\begin{array}{lcll} C & ::= & \alpha \mid \sigma \mid \beta \mid \kappa \mid \chi & \text{component} \\ \kappa & ::= & \mathtt{pre}(x,e,s) \mid \mathtt{post}(x,e,a) \mid \mathtt{emit}(x,e) & \text{state} \\ m & ::= & i : v & \text{message} \end{array}$$

The five kinds of components, of which only κ has a defined structure, are modelled as follows:

- Actuators define a labelled transition system $\langle Set(\alpha), \rightarrow_{\mathbf{a}}, Set(a) \rangle$: a transition $\alpha \xrightarrow{a}_{\mathbf{a}} \alpha'$ models an actuator component (generally made of a set of physical actuators) that moves from state α to α' receiving actuation value a.
- Sensors define a labelled transition system $\langle Set(\sigma), \rightarrow_{\mathbf{s}}, Set(s) \rangle$: a transition $\sigma \xrightarrow{s}_{\mathbf{s}} \sigma'$ models a sensor component (generally made of a set of physical sensors) that moves from state σ to σ' emitting sensor value s.
- Behavior component β is a pure function of the kind $\beta(x,e,\sigma) = (x', e', \alpha)$, namely it consumes a state, input export, and sensed value, producing a new state, output export, and actuation values.
- Communication interfaces define a labelled transition system $\langle Set(\chi), \rightarrow_{\mathbf{c}}, (Set(e) \cup Set(m)) \times \{\uparrow, \downarrow\} \rangle$: a transition $\chi \xrightarrow{\downarrow e}_{\mathbf{c}} \chi'$ models a communication interface component that moves from state χ to χ' internally receiving export e, $\chi \xrightarrow{\uparrow e}_{\mathbf{c}} \chi'$ internally producing export e, $\chi \xrightarrow{\downarrow m}_{\mathbf{c}} \chi'$ receiving message m from outside (namely from another device), and finally $\chi \xrightarrow{\uparrow m}_{\mathbf{c}} \chi'$ sending message m outward—note that an export is a general representation of neighbor values (either imported

or exported), and the interface component is in charge of turning it into a set of messages for neighbors, and vice-versa.

- State component κ is as described in the grammar: it keeps track of the status of a component, of the information useful in that status, and evolves as detailed by the rules in the following.

The semantics of a whole logical device (also called a node in this semantics) can be defined in terms of the behaviors and interactions of its logical components, which we associate to node identifiers to form *situated components*. Interactions between two such situated components can produce an observable action o:

$$\begin{array}{rcll} S & ::= & \langle i, C \rangle & \text{situated component} \\ l & ::= & \mathtt{act}(a) \,|\, \mathtt{sense}(s) \,|\, \mathtt{put}(e) \,|\, \mathtt{get}(e) \,|\, \mathtt{comm}(v) \,|\, \mathtt{compute}(e,a) & \text{action label} \\ o & ::= & i \rhd l \rhd i & \text{observable action} \end{array}$$

The semantics of such interaction is given by a labelled transition system $\langle Set(S) \times Set(S), \rightarrow_{\mathbf{N}}, Set(o) \rangle$, where $\langle S_1 : S_2 \rangle \xrightarrow{i \rhd l \rhd i'}_{\mathbf{N}} \langle S_1' : S_2' \rangle$ is used to mean the two situated components S_1, S_2 interact by action l from device i to i' moving to S_1', S_2'. This is defined by the following rules:

$$\begin{array}{rcll} \langle i, \sigma \rangle : \langle i, \mathtt{pre}(x,e,s) \rangle & \xrightarrow{i \rhd \mathtt{sense}(s') \rhd i}_{\mathbf{N}} & \langle i, \sigma' \rangle : \langle i, \mathtt{pre}(x,e,s') \rangle & \text{if } \sigma \xrightarrow{s'}_{\mathbf{s}} \sigma' \\ \langle i, \chi \rangle : \langle i, \mathtt{pre}(x,e,s) \rangle & \xrightarrow{i \rhd \mathtt{put}(e') \rhd i}_{\mathbf{N}} & \langle i, \chi' \rangle : \langle i, \mathtt{pre}(x,e',s) \rangle & \text{if } \chi \xrightarrow{\uparrow e'}_{\mathbf{c}} \chi' \\ \langle i, \beta \rangle : \langle i, \mathtt{pre}(x,e,s) \rangle & \xrightarrow{i \rhd \mathtt{compute}(e',a) \rhd i}_{\mathbf{N}} & \langle i, \beta \rangle : \langle i, \mathtt{post}(x',e',a) \rangle & \text{if } \beta(x,e,s) = (x',e',a) \\ \langle i, \alpha \rangle : \langle i, \mathtt{post}(x,e,a) \rangle & \xrightarrow{i \rhd \mathtt{act}(a) \rhd i}_{\mathbf{N}} & \langle i, \alpha' \rangle : \langle i, \mathtt{emit}(x,e) \rangle & \text{if } \alpha \xrightarrow{a}_{\mathbf{a}} \alpha' \\ \langle i, \chi \rangle : \langle i, \mathtt{emit}(x,e) \rangle & \xrightarrow{i \rhd \mathtt{get}(e) \rhd i}_{\mathbf{N}} & \langle i, \chi' \rangle : \langle i, \mathtt{pre}(x, e_\perp, s_\perp) \rangle & \text{if } \chi \xrightarrow{\downarrow e}_{\mathbf{c}} \chi' \\ \langle i, \chi_a \rangle : \langle i', \chi_b \rangle & \xrightarrow{i \rhd \mathtt{comm}(v) \rhd i'}_{\mathbf{N}} & \langle i, \chi_a' \rangle : \langle i', \chi_b' \rangle & \text{if } \chi_a \xrightarrow{\uparrow i' : v}_{\mathbf{c}} \chi_a',\ \chi_b \xrightarrow{\downarrow i : v}_{\mathbf{c}} \chi_b' \end{array}$$

That is, the node acquires context information via σ and χ in status `pre`, then computes according to its behavior β, moving to status `post`, performs actuations via α, moving to status `emit` where it finally loads the export into χ, going back to status `pre` for another cycle. This kind of loop provides a proper pace for self-organization through MAPE-based "rounds" involving context monitoring, reasoning, and action. Notice that action $i \rhd \mathtt{comm}(v) \rhd i'$ is the only one involving two distinct devices i and i'; in other words, all the other actions are logically local to a given device.

3.1.2. Formalization: Logical Network Semantics

On top of the formalization of interactions between pairs of situated components as presented in the previous subsection, we now define the model of a whole logical network, starting from the following grammar:

$$\begin{array}{rcll} N & ::= & 0 \,|\, C \,|\, (N|N) & \text{node configuration} \\ L & ::= & 0 \,|\, \langle i, N \rangle \,|\, (L|L) & \text{logical network} \end{array}$$

A node configuration N is defined as a composition of components C by operator "|", and a logical network is defined as a composition of terms of the kind $\langle i, N \rangle$, describing a configuration N situated at a logical device with identifier i. As common in process algebraic approaches [63], we find it useful to define the network semantics first by introducing a congruence relation (i.e., an equivalence relation applicable at any level of depth), equating network configurations defined to be equivalent. This is used to define symbol "|" as multiset composition operator (associative, commutative, and absorbing configuration 0), and to allow blocks $\langle i, N \rangle$ to freely break by splitting N; namely:

$$N|N' \equiv N'|N, \quad 0|N \equiv N, \quad N|(N'|N'') \equiv (N|N')|N''$$
$$L|L' \equiv L'|L, \quad 0|L \equiv L, \quad L|(L'|L'') \equiv (L|L')|L''$$
$$\langle i, N|N' \rangle \equiv \langle i, N \rangle | \langle i, N' \rangle, \quad \langle i, 0 \rangle \equiv 0$$

We then introduce a non-deterministic operational semantics for logical networks by the transition system $\langle Set(L), \rightarrow_{\mathbf{L}}, Set(o) \rangle$. As with transition $\rightarrow_{\mathbf{N}}$ described in Section 3.1.1, we write $L \xrightarrow{i \triangleright l \triangleright i'}_{\mathbf{L}} L'$ to mean that logical network L moves to L' by executing action l from device i to i'. As already discussed, actions of the kind $i \triangleright l \triangleright i$ represent interaction localized inside device i, while $i \triangleright l \triangleright i'$ with $i \neq i'$ represents an interaction through the neighboring link connecting i to i'. The transition relation is defined by the following rules:

$$\begin{array}{lrcll}
\texttt{(L-CONG)} & L_a & \xrightarrow{o}_{\mathbf{L}} & L_b & \text{if} \quad L_a \equiv L'_a, \quad L'_a \xrightarrow{o}_{\mathbf{L}} L'_b, \quad L_b \equiv L'_b \\
\texttt{(L-PAR)} & L_a|L & \xrightarrow{o}_{\mathbf{L}} & L'_a|L & \text{if} \quad L_a \xrightarrow{o}_{\mathbf{L}} L'_a \\
\texttt{(L-INT)} & L_a|L_b & \xrightarrow{o}_{\mathbf{L}} & L'_a|L'_b & \text{if} \quad L_a : L_b \xrightarrow{o}_{\mathbf{N}} L'_a : L'_b
\end{array}$$

The former two rules are standard: the first simply states that transitions work modulo the congruence relation; the second states that transitions can be applied to a network sub-part. The key rule is the third one: it selects two logical network pieces L_a and L_b which can interact by transition $\rightarrow_{\mathbf{N}}$ and action o, and then derives their new states L'_a and L'_b.

A logical network L is said *complete* if it is congruent to a network of the kind:

$$L \equiv \langle i_1, \alpha_1 | \sigma_1 | \chi_1 | \kappa_1 | \beta_1 \rangle | \langle i_2, \alpha_2 | \sigma_2 | \chi_2 | \kappa_2 | \beta_2 \rangle | \ldots | \langle i_n, \alpha_n | \sigma_n | \chi_n | \kappa_n | \beta_n \rangle$$

namely if every logical device i_k in the network has associated precisely five components, one per kind: α_k, σ_k, χ_k, κ_k, and β_k. A network evolution is then defined as a sequence of transitions of the kind:

$$L_0 \xrightarrow{o_0}_{\mathbf{L}} L_1 \xrightarrow{o_1}_{\mathbf{L}} L_2 \xrightarrow{o_2}_{\mathbf{L}} \ldots$$

where L_0 is complete. It is trivially shown that all L_i are complete, since all rules of the transition system leaves the location of components unchanged.

3.2. Platform System and Deployment as a Cyber-Physical Mapping

We represent the platform as a collection of physical *hosts*, connected by a possibly dynamic graph of *physical network links*, representing communication channels connecting a source host with a target host. A host is an entity with a *network identity* (e.g., a URI resource or an IP address); it can be, e.g., a computer system, a device holding a sensor, an actuator, a virtual machine, or a software container. The actual nature of the communication channel may also vary as well, depending on the network architecture and protocols. The sole relevant property is the availability of a directional channel allowing for application-level communication. Additionally, we distinguish: *thin* hosts, which are resource-constrained and may host sensors/actuators but not computations; from *thick* hosts, which instead can compute and may even do so on behalf of multiple logical devices.

We then define a deployment as an *allocation map* placing logical components of each device to specific hosts in the platform system, the arranging platform-level connections to support expected logical connections. An example of a deployment is pictorially shown in Figure 3. For simplicity in presentation, we assume that all the sensors and all the actuators of a device are deployed together (into a single σ- and α-component), though individual sensors and actuators may potentially be placed to different hosts. In Figure 4, we provide examples of *notable deployments*, showing a progression where increasing numbers of responsibilities are centralized (e.g., to a cloud or fog service layer). Of course, depending on the specifics of the actual system being deployed, some cyber-physical mappings may not be actually supported by the platform. For instance, in sensor networks whose sensing devices are

designed to operate for a long time using solely battery power, such devices may not be equipped with enough computation power to host a β-component, making the deployment depicted in Figure 4a practically unfeasible.

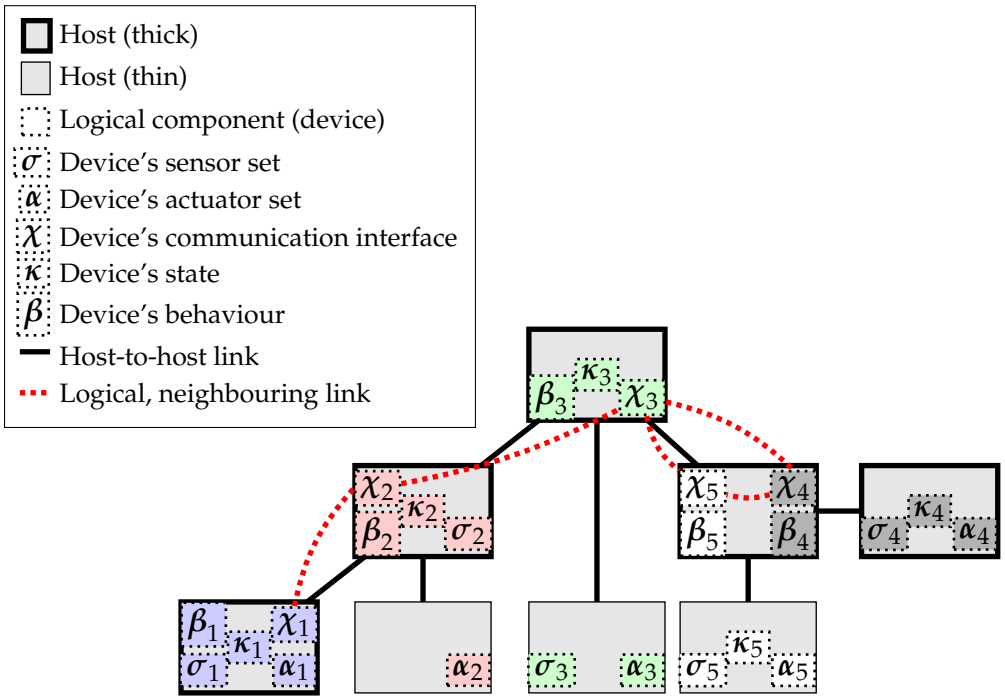


Figure 3. Example instantiation of the CPS model. Dotted graphical elements denote logical components/connections, while solid-line elements denote platform or physical components/connections. Different subscripts (or colors) map to different logical devices. Intra-device connections (Figure 2) are not shown but they can cross host boundaries by following direct host-level links.

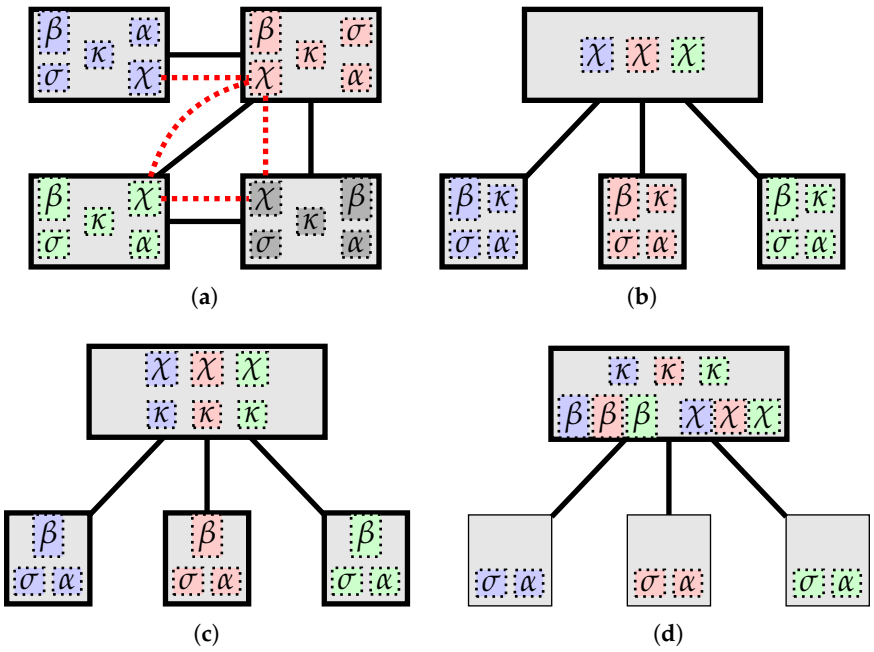


Figure 4. Examples of notable deployments. Refer to Figure 3 for a legend of the graphical symbols. (**a**) Peer-to-peer style; (**b**) Broker-based, as found in IoT publish/subscribe protocols (e.g., MQTT); (**c**) Big data in the cloud; (**d**) Thin hosts with only sensors/actuators.

Details on *how* to perform the mapping of a dynamic logical system upon a changing network of hosts go beyond the scope of this work. Possible approaches from literature are, e.g., in the context of *virtual network embedding* [65]. A further opportunity enabled by the proposed model is *self-(re)configuration*: i.e., the autonomous process by which the platform system (re)arranges components and connections into new deployment configurations. This is a very interesting and important direction for future work.

3.2.1. Formalization: Deployed Network Semantics

We now incrementally formalize the semantics of a network deployed on a platform system. Let meta-variable h range over host identifiers (uniquely identifying physical or virtual devices being targets for deployment). A deployed network is a composition of terms $[h]L$, modelling a (complete or incomplete) logical network L located at a host with identifier h. Although evolving due to the interaction of logical components, deployed networks will then exhibit so-called *global labels*, providing information on both the logical and physical links. As in previous sections, we introduce syntax

$$\begin{array}{rcll} D & ::= & 0 \mid [h]L \mid (D|D) & \text{deployed network} \\ g & ::= & (h,i) \rhd l \rhd (h,i) & \text{global labels} \end{array}$$

and corresponding congruence:

$$D|D' \equiv D'|D, \quad 0|D \equiv D, \quad D|(D'|D'') \equiv (D|D')|D''$$
$$[h](L|L') \equiv [h]L|[h]L', \quad [h]0 \equiv 0$$

We then introduce a non-deterministic operational semantics for deployed networks, by the transition system $\langle Set(D), \rightarrow_{\mathbf{D}}, Set(g)\rangle$. Write $D \xrightarrow{(h,i)\rhd l\rhd(h',i')}_{\mathbf{D}} D'$ to mean that deployed network D moves to D' by executing action l from device i at host h to i' at host h'—when h and h' are the same it means this is an action internal to a host, otherwise it pertains communication across a network link. The transition relation is defined by the following rules:

$$\begin{array}{lrcll} \texttt{(D-CONG)} & D_a & \xrightarrow{g}_{\mathbf{D}} & D_b & \text{if} \quad D_a \equiv D'_a, \quad D'_a \xrightarrow{g}_{\mathbf{D}} D'_b, \quad D_b \equiv D'_b \\ \texttt{(D-PAR)} & D_a|D & \xrightarrow{g}_{\mathbf{D}} & D'_a|D & \text{if} \quad D_a \xrightarrow{l}_{\mathbf{D}} D'_a \\ \texttt{(D-INT)} & [h_a]L_a|[h_b]L_b & \xrightarrow{(h_a,i_a)\rhd l\rhd(h_b,i_b)}_{\mathbf{D}} & [h_a]L'_a|[h_b]L'_b & \text{if} \quad L_a : L_b \xrightarrow{i_a\rhd l\rhd i_b}_{\mathbf{N}} L'_a : L'_b \end{array}$$

Again, while the former two are standard, the third is the key one, capturing the 1-to-1 correspondence with logical network evolution.

3.2.2. Formalization: Deployment

Given a deployed network D, its corresponding logical network $\mathcal{L}(D) \in Set(L)$ can be simply obtained by erasing all sub-terms $[h]L$ to L, namely:

$$\mathcal{L}(D|[h]L) := \mathcal{L}(D)|L \qquad \mathcal{L}(0) := 0$$

We say that a deployed network D is complete if its logical network $\mathcal{L}(D)$ is complete. We analogously define with abuse of notation $\mathcal{L}(g) \in Set(o)$ to be the observable (logic) action obtained from global action g erasing information about physical hosts, namely:

$$\mathcal{L}((h_a,i_a) \rhd l \rhd (h_b,i_b)) := i_a \rhd l \rhd i_b$$

Definition 1. *Define a deployment mapping any function* $\mu : Set(L) \mapsto Set(D)$ *such that* (i) $\mu(L|L') = \mu(L)|\mu(L')$ *and* (ii) $\mu(\langle i, N\rangle) = [h]\langle i, N\rangle$ *for some h. Please note that any deployment mapping* μ *is such that for any logical network L then* $\mathcal{L}(\mu(L)) = L$.

Then, the following correspondence result holds.

Theorem 1. *Cyber-Physical correspondence. Given a complete deployed network* D_0*, for any evolution trace*

$$D_0 \xrightarrow{g_0}_{\boldsymbol{D}} D_1 \xrightarrow{g_1}_{\boldsymbol{D}} D_2 \dots$$

there exists an equivalent evolution trace in the corresponding logical network

$$L_0 \xrightarrow{o_0}_{\boldsymbol{L}} L_1 \xrightarrow{o_1}_{\boldsymbol{L}} L_2 \dots$$

and vice-versa, such that for all k, $L_k = \mathcal{L}(D_k)$ *and* $o_k = \mathcal{L}(g_k)$.

Proof. The thesis is straightforwardly proved by the way $\rightarrow_{\mathbf{L}}$ and $\rightarrow_{\mathbf{D}}$ have been constructed, noting that rules `D-INT` and `L-INT` share the same premises, and hence, if

$$D|[h_a]L_a|[h_b]L_b \xrightarrow{(h_a,i_a)\triangleright l\triangleright(h_b,i_b)}_{\mathbf{D}} D|[h_a]L'_a|[h_b]L'_b$$

then

$$\mathcal{L}(D|[h_a]L_a|[h_b]L_b) \xrightarrow{\mathcal{L}((h_a,i_a)\triangleright l\triangleright(h_b,i_b))}_{\mathbf{L}} \mathcal{L}(D|[h_a]L'_a|[h_b]L'_b)$$

and vice-versa. □

Lemma 1. *Deployment independence. Given a logical network L and deployment mappings* μ_a *and* μ_b*, then the deployed networks* $D^a = \mu_a(L)$ *and* $D^b = \mu_b(L)$ *are such that for any evolution trace*

$$D^a_0 \xrightarrow{g^a_0}_{\boldsymbol{D}} D^a_1 \xrightarrow{g^a_1}_{\boldsymbol{D}} D^a_2 \dots$$

there exists an equivalent evolution trace

$$D^b_0 \xrightarrow{g^b_0}_{\boldsymbol{D}} D^b_1 \xrightarrow{g^b_1}_{\boldsymbol{D}} D^b_2 \dots$$

and vice-versa, such that the corresponding logical evolutions are the same, namely for all k, $\mathcal{L}(D^a_k) = \mathcal{L}(D^b_k)$ *and* $\mathcal{L}(g^a_k) = \mathcal{L}(g^b_k)$.

Proof. Straightforward. □

In this framework, the examples of mappings from Figure 4 can be formalized as follows:

- *Peer-to-peer (Figure 4a).* There is a bijection $i \mapsto h_i$ such that for each C, $\mu(\langle i, N\rangle) = [h_i]\langle i, N\rangle$.
- *Broker-based (Figure 4b).* There is a bijection $i \mapsto h_i$ and a h^b different from all h_i, called a *broker*, such that: $\mu(\langle i, \chi\rangle) = [h^b]\langle i, \chi\rangle$ and for each $C \notin Set(\chi)$, $\mu(\langle i, C\rangle) = [h_i]\langle i, C\rangle$.
- *State as Big Data (Figure 4c).* There is a bijection $i \mapsto h_i$ and a h^c different from all h_i, called e.g., the *cloud*, such that: $\mu(\langle i, \chi\rangle) = [h^c]\langle i, \chi\rangle$, $\mu(\langle i, \kappa\rangle) = [h^c]\langle i, \kappa\rangle$, and for each $C \notin Set(\chi) \cup Set(\kappa)$, $\mu(\langle i, C\rangle) = [h_i]\langle i, C\rangle$.
- *Cloud and thin hosts (Figure 4d).* There is a bijection $i \mapsto h_i$ and a h^c different from all h_i, called e.g., the *cloud*, such that: $\mu(\langle i, \chi\rangle) = [h^c]\langle i, \chi\rangle$, $\mu(\langle i, \kappa\rangle) = [h^c]\langle i, \kappa\rangle$, $\mu(\langle i, \beta\rangle) = [h^c]\langle i, \beta\rangle$, and for each $C \notin Set(\chi) \cup Set(\kappa) \cup Set(\beta)$, $\mu(\langle i, C\rangle) = [h_i]\langle i, C\rangle$.

4. Application to Aggregate Computing

Aggregate Computing [15,66] is a programming paradigm arising from the general need for capturing, linguistically and computationally, the self-organizing behavior of a (possibly large-scale, dynamic) collection of components. Its core idea is to shift from the traditional *device-centric* viewpoint, which focusses on the behavior of each individual agent, to an *aggregate-centric* viewpoint that emphasizes the global behavior of a *collective* or *aggregate system* (i.e., a whole set of interacting autonomous entities). This stance is not just conceptual, but also pragmatical: the collective logic can be expressed as an *aggregate program* that is executed by an *"aggregate virtual machine"* consisting of an entire (possibly distributed) system of networked agents. Aggregate programs are written in some *aggregate programming Domain-Specific Language (DSL)* such as the standalone DSL *Protelis* [16] or the Scala-internal DSL *ScaFi* [17].

In this paper, we do not extend the aggregate computing model itself, but rather describe it as an instance of the model outlined in Section 3, as follows. A logical aggregate system is a cyber system, and its deployment is an allocation map between its component and a platform system. An aggregate cyber system comprises the following elements:

- *Global program*. All devices run the same program β. In aggregate computing, it is possible to express collective behavior through a single, global program which yields different results and actions when *interpreted* against different contexts. Notice that other macro-programming languages [67] adopt a different approach where local programs for the individual nodes are obtained by *compilation* of a global program, so leading to multiple βs.
- *Context* and *Local computation*. The same program is meant to be *iteratively* and *asynchronously* evaluated *locally* against each device's up-to-date context, which consists of a tuple extracted from κ with (i) a map of the most recent sensor values for the named sensors in σ used by the program itself, and (ii) a map from neighbors to their most recent exported value. That is, only the last message for each neighbor is to be stored. Moreover, messages are usually retained for an application-specific amount of time. This suffices to guarantee progression and steering (via self-organization).
- *Program output*. According to the aggregate operational semantics [64], the output of a local execution of an aggregate program is a *vtree*, i.e., a tree of values. A vtree already embeds both (i) local state to support stateful computations; (ii) information needed for coordination, to be broadcast to neighbors; and (iii) prescriptions about actuations to be performed by α. In other words, the output of an aggregate program is directly usable as an *export* (see Section 3.1.1).
- *Messages*. Messages sent to neighbors through χ components are (specific parts of) vtrees.

Details about *how* compositions of aggregate computing constructs give rise to self-organizing behavior are deeply covered in [64]. It suffices to say that aggregate programs can be defined as *compositions* of functional blocks that describe how a system should collectively behave through a manipulation of *computational fields* [64], i.e., dynamic maps from logical devices to computational values. However, each device computes against partial fields given by the most recent information gathered from neighbors. The key insight delivered in this paper is that the execution of such aggregate programs can be "pulverized" into micro-steps of sensing, computation, interaction, and actuation (Section 3.1) that can also be distributed across different machines. Consider the example of a *self-healing channel* [20]: an algorithm that yields, for each device, a Boolean value denoting whether it is part of the shortest path from a source to a destination group of devices, expressed in Protelis as shown in Figure 5.

```
def channel(source, destination, tolerance) {
  distanceTo(source) + distanceTo(destination)  <=
    distanceBetween(source, dest) + tolerance
}
```

Figure 5. A self-healing channel in Protelis.

There, `distanceTo(s)` is a gradient (a field of minimum distances to source devices for which field `s` holds `true`) and `distanceBetween(s,d)` a function computing and propagating everywhere the distance between `s` and `d`. When this specification is executed in a decentralized, iterative way, it locally adapts the channel value after changes in topology, device-to-neighbor distances (obtained through a proper sensor used within `distanceTo`—not shown), and input fields `source`, `destination`, and `tolerance`. Notice how the specification is declarative: it describes the macro-level behavior without specifying *exactly* how micro-level behavior is to be carried out. This enables pulverization of the system, first into devices (and their execution steps) and then into devices' components (and their execution steps). Pulverization does not depend on the specific program: the execution protocol is "fixed"; only communication payloads, which include vtrees yielded by local evaluations of the program, change. Each aggregate function call (such as `distanceTo`) yields a particular sub-tree including its very own relevant data.

5. Deployment Independence in Action: Aggregate Computing over DingNet

In this section, we leverage deployment independence to integrate aggregate programming, embodied by the Protelis [16] programming language, into DingNet [23,68], an exemplar for research on self-adaptation in the domain of IoT.

DingNet provides an integrated simulator that maps directly to a physical IoT system deployed in the area of Leuven, Belgium. Such IoT deployment includes a situated Low-Power Wide-Area Network (LPWAN) realized with LoRaWAN devices [13], whose hardware does not allow for intensive computation, making them obligated *thin hosts*. We extend the simulator adding support for simulating edge servers and non-LoRaWAN situated devices, thus enabling controlled experiments involving a heterogeneous network composed of a mixture of thick and thin nodes whose communication intertwines several network protocols. We show how the proposed approach enables aggregate computations to be performed over such a heterogeneous infrastructure, abstracting away deployment its complexity from the business logic design.

5.1. LoRaWAN

We here briefly introduce LoRaWAN and its software stack. LoRaWAN is a communication system designed for long-range communication (up to kilometers) and low energy requirements, at the price of data rates in the order of few bytes per second (six or more orders of magnitude less than Wi-Fi or LTE). The foundational elements of LoRaWANs are end devices (or motes), gateways, and LoRa servers. Motes communicate with gateways, forming a star topology. Gateways, in turn, communicate with a LoRa server, building an overall star of stars topology. In such a network, each mote willing to communicate broadcasts its message to all nearby gateways through single hub wireless links. All the gateways receiving the packet forward it to the server through any available networking backhaul (the protocol does not govern how gateways server communicate). When communicating from servers to motes, the server selects the gateway that deems best to deliver data to the recipient. LoRa motes are designed for constrained-energy scenarios—typical industrial implementations provide battery life of 1–3 years. Consequently, motes provide little computation capacity and memory, and hence represent an exemplar case of *thin* devices.

5.2. MQTT-Based Network Architecture

Even though the LoRaWAN protocol focuses on communication between motes and gateways, its typical implementations usually resort to a centralized cloud LoRa server (most notably, relying on *The Things Network* (https://www.thethingsnetwork.org/) [69]), or the *Chirpstack* implementation (https://www.chirpstack.io/)) , which provides support for communicating with motes through high-level protocols, typically MQTT [21]. MQTT is a connectivity protocol designed as a lightweight publish/subscribe messaging transport devoted to IoT and machine-to-machine communication. It defines two network entities: clients that want to exchange information; and a broker, which is in charge of receiving and redistributing data among clients. As part of the contribution of this work, we extended DingNet with an explicit model of the interaction between gateways and LoRa servers via MQTT, thus emulating the behavior of Chirpstack. The implementation has been released as open source under a permissive licence (https://github.com/aPlacuzzi/DingNet/releases/tag/v1.3.3).

5.3. Aggregate Computing over DingNet

We now define our cyber system and the mapping of pulverized components with the platform layer given by DingNet.

The cyber system, which we target when defining the business logic, is composed of several logical devices with computation, sensing, actuation, and communication capabilities. Logical components have a position in space, and their neighborhoods are defined based on geographical proximity, within some arbitrarily chosen range (although, in principle, any policy for neighborhood definition could be used). The set of sensors and actuators may differ on each device. All logical devices have computational capabilities. Every logical device is *bound* to a situated platform device of the simulated system, inheriting its spatial situation.

Table 1. Available choices for pulverized component deployment for different device types in the extended DingNet simulator. D indicates that the component can be deployed on the local device directly, E that it can get deployed on edge, and C on cloud. The simulator supports only configurations where behavior and state are located onto the same type of host (e.g., edge-edge). The devices can be *thin* (e.g., LoRa mote) or *thick* hosts (e.g., handhelds).

Device Type	Sensors (σ)	Actuators (α)	Behavior (β)	State (κ)	Communication (χ)
Thin host	D	D	E/C	E/C	E /C
Thick host	D	D	D/E/C	D/E/C	E/C

The pulverized logical system is then mapped over the platform system as follows (see also Table 1). Sensing and actuation systems σ and α are always deployed on the host bound to the logical node. This is true for both *thin* and *thick* hosts. Behavior (β) and state (κ) components cannot get allocated on thin hosts: their deployment targets are restricted to the cloud, edge servers, and thick situated devices. To simulate realistic situations, we allow β and κ components of LoRa motes associated devices to be located solely on edge servers or on the cloud, and we let β and κ associated with localized thick devices (e.g., handhelds) to be hosted locally or on the cloud. Impractical deployments, (e.g., β and κ on a device backing the LoRa mote [70]) although possible in principle, are deemed as unrealistic with the current state-of-the-art technologies, and thus excluded from the present analysis. Moreover, as introduced in Section 4, we note that all logical devices share the same behavior: $\beta_i = \beta_j \; \forall \; i, j$ where i, j are logical device identifiers. This is part of the semantics of aggregate computing, and may vary for other pulverizable computational models. We give an example program of logical devices for the evaluation case in Section 6. Finally, communication components χ are all hosted on a single device, the MQTT broker, that can be either an edge server or a service on the cloud. This choice was made for the sake of simplicity and without loss of generality: in fact, even though in principle χ components could be located on diverse hosts, and communication among them could be wired to any level 7 network protocol, we share in fact the same MQTT broker supporting the

Chirpstack architecture. Figure 6 summarizes the aggregate system as deployable within the DingNet extended simulator.

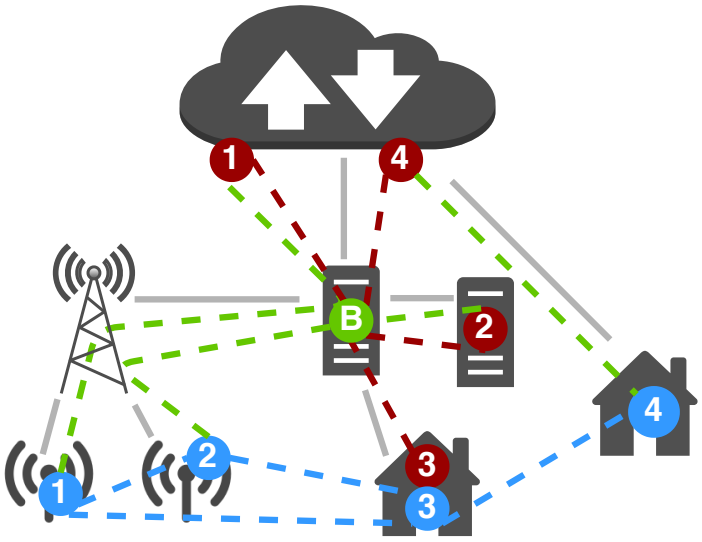


Figure 6. Abstract architecture of the experiment, graphical elements share the semantics of Figure 1c, with some additional elements: red dashed lines represent platform-level MQTT communications; small antennas identify thin hosts (LoRa motes); large antennas are LoRa gateways, houses depict situated thick devices, and vertical gray rectangles symbolize edge servers. The green circle with a B marks a MQTT broker, which can be deployed on edge servers as in the figure, or in the cloud.

6. Evaluation

We exercise our approach for deployment independence via simulation, relying on the aforementioned extended DingNet simulator with Protelis support. In particular, we simulate a CPS whose aim is to reduce the contribution of household winter heating to air pollution [71] by imposing custom maximum household temperatures relative to the current level of particulate matter with a diameter smaller than 10 µm (PM_{10}) in the area surrounding the household.

The system must provide this functionality in a self-organizing fashion: there must be no central coordinator; rather, the system itself must autonomously organize its micro-level actuation activity, in terms of decentralized computation and interaction, so that the functionality is implemented achieved despite environmental perturbations. Our goal is to show that the same business logic, defined once, can be reused via its pulverized model across different deployment schemes, allowing the picking of an actual deployment whose non-functional tradeoffs better fit the requirements at hand.

The whole experiment has been automated, documented, and open sourced in a public repository (https://github.com/aPlacuzzi/Experiment-2020-FutureInternet-LoRa/) to facilitate accessibility and reproduction.

6.1. Case Description and Simulation Configuration

Our CPS is composed of a sensor network of LoRa motes, each equipped with a sensor measuring PM_{10} in µg/m³. Eight motes are installed in fixed position, while two are mobile (mounted on public transport systems). LoRa network coverage on the city is provided by nine gateways, whose position in the simulated environment matches one of the physical gateways deployed in the Belgian city of Leuven. The target of the control system are 300 households, randomly selected in the city area, equipped with a smart thermostat. Our logical neighborhood relationship allows logical direct communication in a unit disc of 1km radius.

The deployment possibilities and network architecture are summarized in Table 2. The actual deployment scheme is controlled by three parameters described in Table 3, indicating respectively: the fraction of LoRa devices whose computation is hosted on the edge rather than on the cloud (P_e);

the fraction of smart thermostats whose computation is hosted on the local device rather than on the cloud (P_l); and whether the MQTT broker is hosted on the edge or on the cloud (B). Changing the values of such parameters generates a new deployment, with its own performance and price tradeoffs: we exercise our approach by showing that the business logic is unaffected by such change.

Table 2. Deployment configurations under simulation. D indicates that the component can be deployed on the local device directly, E that it can get deployed on edge, and C on cloud. Behavior (σ) and state (κ) components must be located onto the same host (device-device, edge-edge, cloud-cloud).

Device Type	Sensors (σ)	Actuators (α)	Behavior (β)	State (κ)	Communication (χ)
MQTT broker on edge server					
LoRa mote	D	D	E/C	E/C	broker's host
Thermostat	D	D	D/C	D/C	broker's host
MQTT broker on cloud service					
LoRa mote	D	D	E/C	E/C	broker's host
Thermostat	D	D	D/C	D/C	broker's host

Table 3. Free variables for the case.

Name	Values	Unit	Meaning
seed	[1, ..., 10]	n/a	Simulation random generator seed
P_e	$\lceil \frac{k}{3} \mid k \in [0, ..., 3]$	n/a	Fraction of LoRa devices whose computation is hosted on the edge
P_l	$\lceil \frac{k}{3} \mid k \in [0, ..., 3]$	n/a	Fraction of thermostats whose computation is hosted on the local device
d_{ee}	$10^k \mid k \in \{-1, 0, 1\}$	ms	Network delay in edge-to-edge communication
d_{cc}	{5, 25, 100}	ms	Network delay in cloud-to-cloud communication
d_{ec}	{10, 50, 300, 1000}	ms	Network delay in edge-to-cloud and cloud-to-edge communication
d_a	{50, 150, 500}	ms	Network access delay for smart thermostats
B	{cloud, edge}	n/a	Deployment position of the MQTT broker

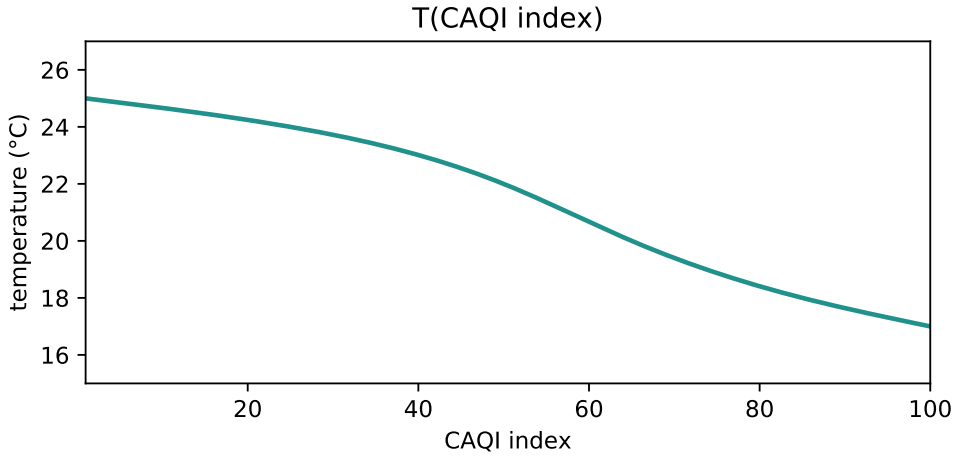


Figure 7. Function used in the case study to set the maximum allowed temperature setpoint based on the current estimation of the CAQI index (a metric of air pollution).

The DingNet simulator takes care of realistically simulating LoRaWAN network interactions and delays, while other communications are simulated with a refined version of the network model proposed by EdgeCloudSim [72]. More precisely, the communication delay (D) is a function of a propagation delay d, the message size s, and the channel data rate b: $D = d + {}^{s}/_{b}$. Since all communications in our system are mediated by an MQTT broker, we consider delays and data rates all relative to communications from and to that broker. In particular, we assume the data rate of the broker to be $1{}^{Gb}/_{s}$, and we compute the propagation delay d considering the actual network path the data needs to travel, summing the individual contributions of each communication link. We account separately for end devices Internet access delay, e.g., due to Wi-Fi or LTE (d_a); communication delays between cloud and edge (d_{ec}), edge servers (d_{ee}), and cloud services (d_{cc}). Actual delay values are free

variables in our experiments, summarized in Table 3. We also take into account the case in which two logically separate components are hosted on the same physical machine, in which case the data rate is considered virtually unlimited, thus $s/b \simeq 0$, and consequently consider such delay as constant $D = d_l = 0.015$ ms.

Logical devices are programmed to: (i) consider pollution data from all nodes within 2 hops; (ii) compute the local pollution value as the weighted mean of such information, using the inverse of the distance as weights; and (iii) use the value to set the maximum allowed temperature, if the appropriate actuator is available. Instead of directly relying on PM_{10} as metric, it computes the CAQI index [73], in such a way that future addition of further pollution metrics such as $PM_{2.5}$ can be considered with no change to behavior. The relationship between the computed CAQI index and the desired temperature used in our experiments is shown in Figure 7.

The program, excerpted in Figure 8, is entirely independent of the actual deployment of logical devices' components.

```
// id, coordinates and CAQI pollution for each neighbour
let neighbors = foldUnion([getUID(),self.getCoordinates(), env.get("CAQI")])
// same information for each neighbour's neighbour
let pollution = distinctByUID(foldUnion(nbr(neighbours)))
    // replace position with distance from myself
    .map { it.set(1, self.distanceTo(it.get(1))) }
// Compute maximum allowed temperature
let maxTemperature = self.temperatureByPollution(pollution)
// If an appropriate actuator exists, set it
if (env.has("thermal_control")) {
    env.put("thermal_control_max", maxTemperature)
}
```

Figure 8. Protelis code for the case study.

We simulate the system evolution over four days (96 h). Device round frequency is set to $1/900$ Hz. We perturb the system by reproducing the reduction in PM_{10} caused by a rainfall [74] starting from the northern side of the city after 25 h of simulation, covering the whole city at the 31st simulated hour, and terminating at the 61st hour. Figure 9 shows the system's evolution through snapshots of a simulation run.

Our reference metrics are presented in Table 4. Temperature measures are used to verify that the system is operating nominally, regardless of the deployment configuration. Latency measures are intended as a performance proxy, showcasing that different deployment choices provide different tradeoffs. We also provide an operating cost estimation for the deployment. Our estimation is the following:

- We estimate the pricing for operating the MQTT broker on the cloud by referring to the prices of a prominent operator available at the time of writing (https://archive.is/9g2tk).
- We assume the broker located on the edge to be self-hosted, thus with a negligible upkeep cost.
- We consider the smallest cloud server instances available from one of the major players (https://archive.is/9fg0e). We estimate that each such instance has enough memory to comfortably host about a hundred pulverized smart thermostats.
- For any cloud hosting service, we pick the pricing for the infrastructure located geographically closest to Leuven (in our case, located in Frankfurt).
- We include the electricity pricing to estimate the operating cost for smart thermostats (Arguably, this cost would be charged to the end users, and thus not impact the actual operating cost directly. However, we decided to consider this negative externality in our estimation), to do so we consider them to be implemented using a Raspberry PI Zero [75].

- We consider the Raspberry PIs to have an idle current drain of 65 mA (https://archive.is/Lk0BI) (hence consuming 0.325 W at 5 V), and a load power consumption of 2.7 W (https://www.phoronix.com/scan.php?page=article&item=raspberry-pi-burst&num=3).
- We estimate the lelectricity price to be 0.28€/KWh (https://archive.is/N7BPS), and we use a conversion factor of 1.2$/€ (https://archive.is/lD9ga).

We execute 10 runs of the experiment with a different seed for every combination of the variables in Table 3. Different seeds imply different household positions, users' desired temperature, and smart device power on instant. Data analysis has been performed using Xarray [76] and matplotlib [77]. The complete analysis, available on the aforementioned repository, counts over 350 charts, of which the most relevant are included here.

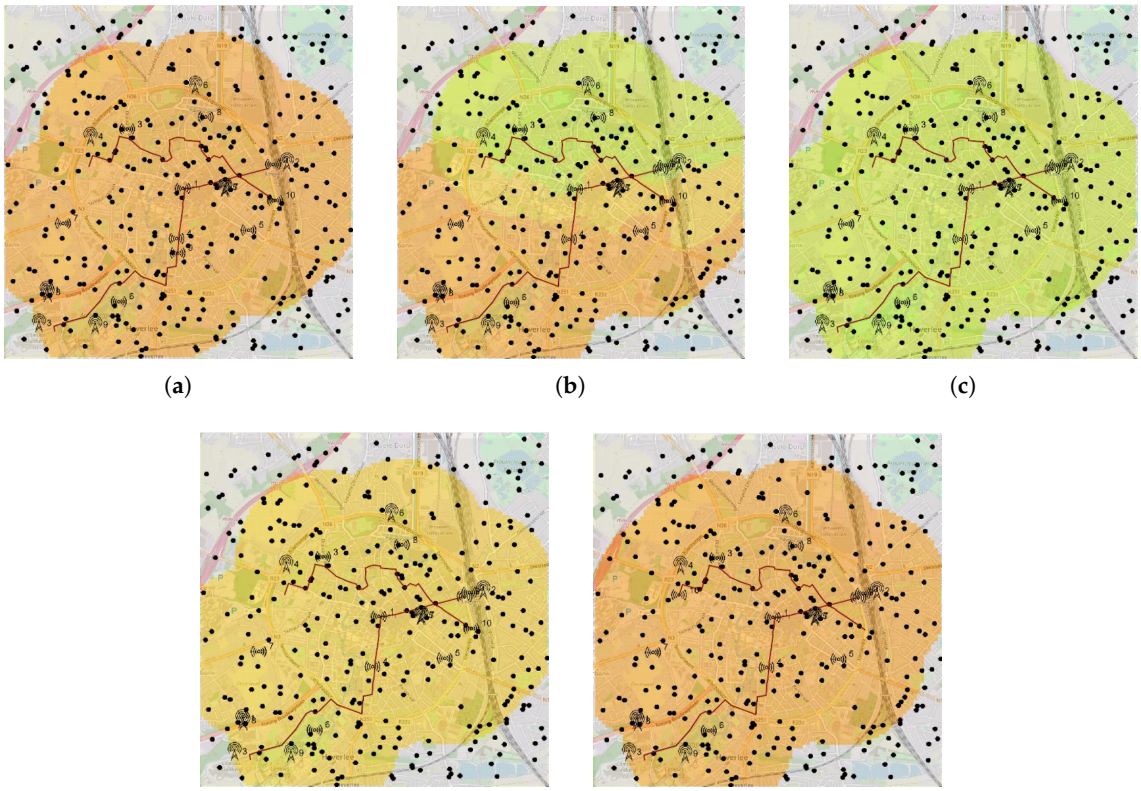


Figure 9. Sequence of snapshots of the simulator during an execution run of the experiment. Initially (**a**), PM_{10} levels are comparable in the whole city. After 25 h from the beginning of the experiment (**b**), intense rain enters from north, lowering the PM_{10} levels, and covers the whole city in six hours (**c**), thus lowering PM_{10} levels everywhere. Rain stops uniformly after 61 h from the beginning of the experiment (**d**), and the overall PM_{10} raises with time to the initial level (**e**).

Table 4. Metrics for the case.

Name	Unit	Meaning
D_{15}	ms	Accumulated network delays in the last 15 min.
T_i	°C	Maximum allowed heating temperature for smart thermostat i
$E[T]$	°C	Mean of T across all devices
$\sigma[T]$	°C	Standard deviation of T across all devices
T_{max}	°C	Maximum of T across all devices
T_{min}	°C	Minimum of T across all devices

6.2. Results and Analysis

Subsequently, we show how the pulverisation approach achieves the functional goals independently of deployment, then we elaborate on the effect of different deployments on non-functional goals, and finally we discuss threats to validity.

6.2.1. Achieving Functional Goals Independently of Deployment

Our goal is to demonstrate that pulverization allows for reusing the same business logic on diverse deployments, thus making deployment no longer a design constraint with respect to the realization of the functionality of the system. Figure 10 shows correctness by depicting the system functionality in a typical configuration. The system, whose overall performance changes in response to different network delays, retains its application-level behavior across the table, regardless of the actual deployment shape of its sub-components; namely the core business logic of the application is preserved regardless of the underlying deployment configuration (whose alternatives are summarized in Table 2). The example thus shows that pulverization achieves separation between the behavior of the system from its deployment detail, thus providing flexibility to engineers that will be able to select (and change) the deployment scheme based on desired performance and projected operating cost.

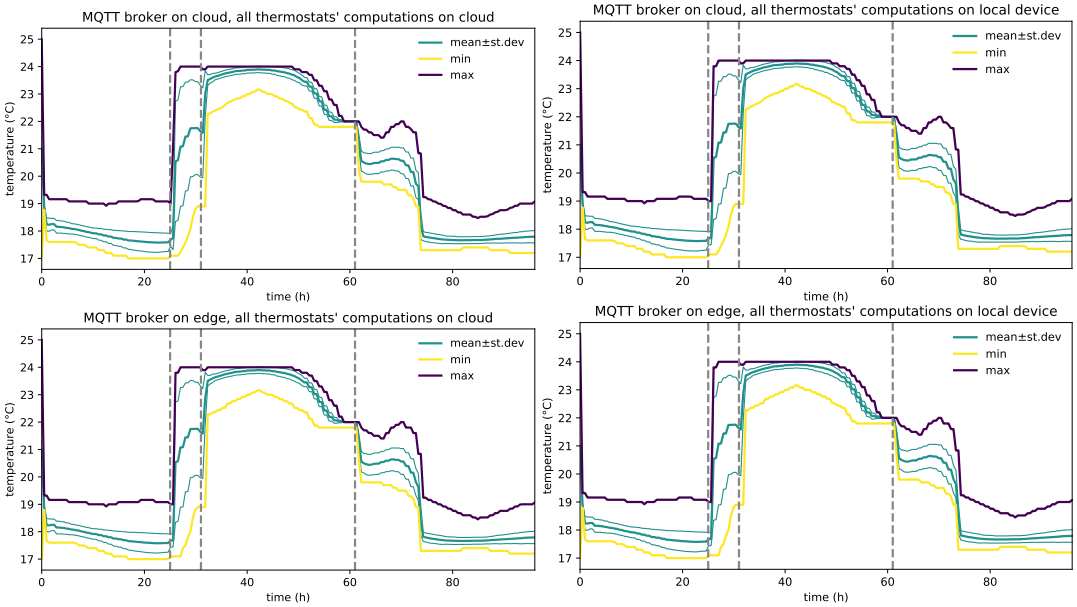


Figure 10. Evolution of $E[T] \pm \sigma[T]$ (aquamarine), T_{max} (purple), and T_{min} (yellow) in the following conditions: B = cloud (MQTT broker hosted on the cloud) in the top charts; B = edge (MQTT broker hosted on the edge), in the bottom charts; $P_l = 0$ (all thermostat computation hosted on the cloud), in the left-hand column; $P_l = 1$ (all thermostat computation hosted on the local device), right-hand column. Every delay is set to the second lowest value in the set (usually resembling common operating conditions). Vertical dashed gray lines depict, from left to right: rainfall arrival on the northern part of the city, rain falling on the total city area, and rainfall conclusion. The functional behavior of the system is consistent regardless of the deployment strategy: results are very similar across the board. Very small differences come from different network performance (mostly due to diverse network latency) and from the randomness inherent to the simulation. The system reacts to changes in the detected PM_{10} level by adjusting the temperature and allowing higher temperatures to be set when air pollution levels are sufficiently low. $\sigma[T]$ increases when rainfall hits only part of the city. This is because thermostats located in the area subject to the rainfall allow higher temperatures to be set, while the remainder of devices remains on the pre-rain levels, thus raising variance.

We stress that no change to the application level software was required to tackle different deployment targets for any component, thus demonstrating the feasibility of a deployment-agnostic approach to self-organizing CPSs design.

6.2.2. Effect of Deployment on Non-Functional Goals

In-depth performance analysis is presented in Figure 11, while cost estimation is depicted in Figure 12. Performance analysis shows that depending on expected network delays, different deployment choices may have relevant performance consequences. Expressing the behavior in a deployment-independent way along with the availability of an evaluation tool enables for predicting the system's performance across several different possibilities, thus allowing for more informed deployment choices. Should conditions change and a different deployment be required to meet performance metrics, application-level logic would not get impacted in any way.

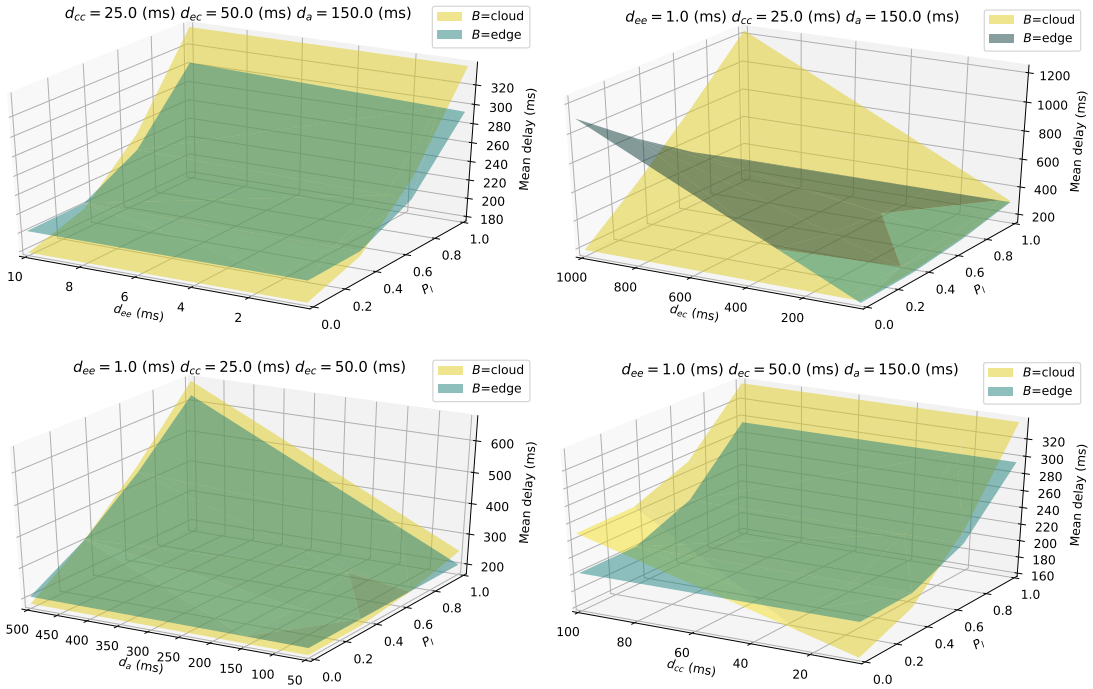


Figure 11. System performance (mean communication delay) for different propagation delays (one per chart) and different deployments, obtained by changing the number of smart thermostats whose code is executed locally (P_l) and the deployment location of the broker (surface color). Lower values indicate better performance. The pulverized system does not require any adjustment to the business logic to tackle the diverse deployment schemes: the designer can thus use the following analysis to decide which deployment configuration to pick depending on how important it is to maximize performance. Changes in edge-to-edge delays (top left) impact performance minimally, as most of the accumulated delay is elsewhere; thus, they provide a good baseline to compare other configurations to. Changing the propagation delay between edge and cloud (top right) highlights that the worse the network performance, the more is important for the broker to be close to where the computation components of the logical system are concretely deployed. Performance of the household connection (bottom left) have higher impact on the overall performance of the CPS with the count of devices being executed locally, regardless of the broker location. This suggests that offloading the business logic to the cloud, thus sending fewer data to end devices, should be considered when the network is expected to experience high load. Finally, a difference in the delay in cloud-to-cloud communications (bottom right) is influential only for cases in which the broker is located there. If there is little or no guarantee on where the cloud components will get allocated (and hence, little knowledge about their communication performance), then adopting an edge-located broker can provide a less volatile performance.

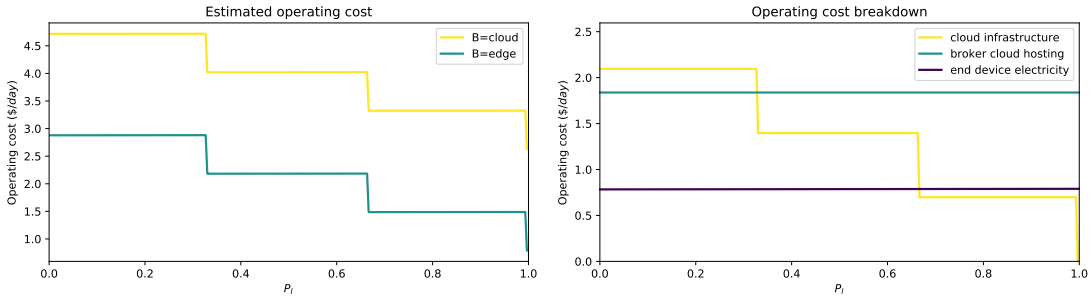


Figure 12. Operating cost estimation for different deployment schemes. *x*-axes of both charts show the fraction of devices being hosted on the local thermostat, as opposed as on the cloud: every point is a deployment scheme that has been exercised retaining functionality, right-hand points are deployments with progressively more component moved to the edge of the network. Pulverization allows for exercising different deployment schemes without any change to the functional specification. Different lines on the eft-hand chart show projected pricing for hosting the broker on the self-hosted edge servers or moving it on the cloud. The right-hand side chart shows a breakdown of the overall pricing into its constituent components. The forecasted usage for the broker does not overcome the first pricing level; it is thus a constant contribution. Idle time power drain dominates the electricity consumption of smart thermostats (although the line looks flat, it imperceptibly grows): this suggests that moving computation towards the edge is generally cheaper, and suggests that better power saving strategies than keeping the device idle should be devised. Cloud hosting pricing decreases when devices are moved to the edge; since we request a new cloud instance every 100 thermostats hosted on cloud and pricing goes per-instance, the chart is discontinuous.

6.3. Threats to Validity

In the section we discuss the main internal (intrinsic to the experiment, e.g., systematic error) and external (related to the generalizability) threats to the validity of the evaluation results, and for each one we explain how we mitigated it.

Regarding the correctness of the application-level behavior in the experiment: changes in the pollution level get reflected in the maximum allowed temperature setpoint for households, and the time after which the change becomes effective is subsequent to the moment sensors detect such change. Guarantees in this sense come from the structure of the application program itself. In fact, it functionally builds the output field (i.e., the field of maximum allowed temperature) through a field computation taking a single input field, namely the field of pollution level. The network performance that we measure as mean network delay, is dependent on all the free variables. For each such variable, we picked a set of values that ranges from smaller-than-normal to larger-than-normal, of course also experimenting with values that could be reasonably expected for a real deployment. To pinpoint the effect of every single variable on the performance, we execute simulation replicas for every combination in the Cartesian product of variable values, hence testing every combination. This allowed us to isolate the behavior and impact of every single change of any variable value, moreover enabling the analysis of interactions among multiple values (e.g., possible resonant phenomena).

Simulations have been performed using state-of-the-art best practices, for instance, multiple randomization-controlled simulation repetitions (ten for each combination of the free variables), guaranteeing both a reasonable problem space exploration and complete reproducibility. The first version of [23] has been used in the past to simulate realistic LoRa communications between LoRa motes and LoRa gateways. Network communications other than those based on the LoRa protocol are modelled using a refined version of the network model proposed in [72], as described in Section 6.1. The most obvious threat to the validity of the network model is the lack of a model of lost packets, along with the assumption that the MQTT broker is behaving correctly and not having networking issues. In case of deployment in networks with low reliability, we expect the deployed system to show performance possibly different from the ones measured in simulation. However, we note that networking among households, edge, and cloud is usually reasonably reliable, and in our simulated

system the most unreliable network is by its nature the wireless LoRa communication, whose model from Dingnet however includes packet losses, collisions, and signal propagation.

Another validity issue concerning generalisability is the resilience of the system in face of an active attacker: this kind of robustness would require specific security measures to be implemented, which fall out of the scope of this manuscript (see, e.g., [78,79]).

At the application level, we note that by its nature LoRa can efficiently deal with systems whose evolution in time is not exceedingly quick. If sensor readings are required multiple times per second, LoRa motes would quickly terminate their allowed use of the shared medium and hence stop transmitting to comply with the medium-use limitations [80], hence affecting the final results. More generally, the applicability of the LoRa network architecture for the specific domain at hand puts some limitations on the generality of the results obtained for this specific case study. For similar applications in different domains, as far as time frames are similar, we expect that the simulation techniques can be straightforwardly applied. We stress, however, that such limitation is due to the specifics of LoRaWAN and affects solely the simulation platform: the concept of pulverization remains untouched, as it is conceptually independent of the network and hardware infrastructure on which we implemented it for the evaluation.

7. Conclusions and Future Work

In this paper, we introduced a novel model for self-organizing cyber-physical systems that fosters "pulverization" of the structure and execution of global system behavior (i.e., the ability to decompose macro-level components and activity into micro-level components and activities) and deployment independence (i.e., the ability of moving components and activities to different deployment targets). We then demonstrated the approach by considering an instantiation in the aggregate computing framework. We validated the approach in a simulation case of pollution-aware household heat monitoring and control, achieved through a simulator extending DingNet to support heterogeneous CPSs comprising LoRa motes, IoT devices, edge servers and Protelis nodes.

As future work, we plan to investigate and develop support for adaptive and opportunistic deployment of self-organizing systems by taking into account dynamics in the environment (for instance based on human behaviour), changes in the available infrastructure (e.g., as induced by failure and mobility), changes in requirements and preferences (e.g.,related to quality of service, operating cost, and risk), and accordingly, dynamic and opportunistic relocation of components. The theoretical framework introduced in this paper will then serve as ground.

Author Contributions: Conceptualization, R.C., D.P., M.V. and D.W.; Data curation, A.P.; Funding acquisition, M.V.; Investigation, R.C.; Methodology, M.V. and D.W.; Software, D.P. and A.P.; Validation, A.P.; Writing—original draft, R.C., D.P. and A.P.; Writing—review and editing, M.V. and D.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the MIUR PRIN Project N. 2017KRC7KT "Fluidware".

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Camazine, S.; Deneubourg, J.; Franks, N.R.; Sneyd, J.; Theraulaz, G. *Self-Organization in Biological Systems*; Princeton Studies in Complexity; Princeton University Press: Princeton, NJ, USA, 2003.
2. Serugendo, G.D.M.; Gleizes, M.P.; Karageorgos, A. Self-organization in multi-agent systems. *Knowl. Eng. Rev.* **2005**, *20*, 165–189. doi:10.1017/s0269888905000494.
3. Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA, 17–21 August 2020. Available online: https://ieeexplore.ieee.org/xpl/conhome/9189891/proceeding (accessed on 17 November 2020).

4. Weyns, D.; Schmerl, B.; Grassi, V.; Malek, S.; Mirandola, R.; Prehofer, C.; Wuttke, J.; Andersson, J.; Giese, H.; Göschka, K.M. On Patterns for Decentralized Control in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II*; de Lemos, R., Giese, H., Müller, H.A., Shaw, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; Volume 7475, pp. 76–107. doi:10.1007/978-3-642-35813-5_4.
5. Hegde, G.; Rao, M. Smart Cloud: A Self-organizing Cloud. In *Inventive Computation Technologies*; Smys, S., Bestak, R., Rocha, Á., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 723–729.
6. Parunak, H.V.D.; Brueckner, S.A. Software engineering for self-organizing systems. *Knowl. Eng. Rev.* **2015**, *30*, 419.
7. Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, Boston, MA, USA, 9–11 July 2007. Available online: https://ieeexplore.ieee.org/xpl/conhome/4274871/proceeding (accessed on 17 November 2020).
8. Proceedings of the 13th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2019, Umea, Sweden, 16–20 June 2019. Available online: https://ieeexplore.ieee.org/xpl/conhome/8765675/proceeding (accessed on 17 November 2020).
9. Brueckner, S.A.; Serugendo, G.D.M.; Karageorgos, A.; Nagpal, R. (Eds.) *Engineering Self-Organising Systems*; Springer: Berlin/Heidelberg, Germany, 2005. doi:10.1007/b136984.
10. Bures, T.; Weyns, D.; Schmerl, B.; Tovar, E.; Boden, E.; Gabor, T.; Gerostathopoulos, I.; Gupta, P.; Kang, E.; Knauss, A.; Patel, P.; Rashid, A.; Ruchkin, I.; Sukkerd, R.; Tsigkanos, C. Software Engineering for Smart Cyber-Physical Systems: Challenges and Promising Solutions. *ACM SIGSOFT Softw. Eng. Notes* **2017**, *42*, 2. doi:10.1145/3089649.3089656.
11. Villari, M.; Fazio, M.; Dustdar, S.; Rana, O.; Jha, D.N.; Ranjan, R. Osmosis: The Osmotic Computing Platform for Microelements in the Cloud, Edge, and Internet of Things. *IEEE Comput.* **2019**, *52*, 14–26. doi:10.1109/MC.2018.2888767.
12. Mekki, K.; Bajic, E.; Chaxel, F.; Meyer, F. Overview of Cellular LPWAN Technologies for IoT Deployment: Sigfox, LoRaWAN, and NB-IoT. In Proceedings of the 2018 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2018, Athens, Greece, 19–23 March 2018; pp. 197–202. doi:10.1109/PERCOMW.2018.8480255.
13. Sornin, N.; Luis, M.; Eirich, T.; Kramp, T.; Hersent, O. LoRaWAN™ Specification. Available online: https://lora-alliance.org/resource-hub/lorawanr-specification-v11 (accessed on 17 November 2020).
14. Wolf, T.D.; Holvoet, T. Designing Self-Organising Emergent Systems based on Information Flows and Feedback-loops. In Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007), Boston, MA, USA, 9–11 July 2007; pp. 295–298. doi:10.1109/SASO.2007.16.
15. Beal, J.; Pianini, D.; Viroli, M. Aggregate Programming for the Internet of Things. *IEEE Comput.* **2015**, *48*, 22–30. doi:10.1109/MC.2015.261.
16. Pianini, D.; Viroli, M.; Beal, J. *Protelis: Practical Aggregate Programming*; ACM SAC: Salamanca, Spain, 2015; pp. 1846–1853. doi:10.1145/2695664.2695913.
17. Viroli, M.; Casadei, R.; Pianini, D. Simulating Large-scale Aggregate MASs with Alchemist and Scala. In Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, Gdansk, Poland, 11–14 September 2016; pp. 1495–1504. doi:10.15439/2016F407.
18. Viroli, M.; Audrito, G.; Beal, J.; Damiani, F.; Pianini, D. Engineering Resilient Collective Adaptive Systems by Self-Stabilisation. *ACM Trans. Model. Comput. Simul.* **2018**, *28*, 1–28. doi:10.1145/3177774.
19. Beal, J.; Viroli, M.; Pianini, D.; Damiani, F. Self-Adaptation to Device Distribution in the Internet of Things. *ACM Trans. Auton. Adapt. Syst.* **2017**, *12*, 12:1–12:29. doi:10.1145/3105758.
20. Viroli, M.; Casadei, R.; Pianini, D. On execution platforms for large-scale aggregate computing. In Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct, Heidelberg, Germany, 12–16 September 2016; pp. 1321–1326. doi:10.1145/2968219.2979129.
21. ISO Central Secretary. *Information technology—Message Queuing Telemetry Transport (MQTT) v3.1.1*; Standard ISO/IEC 20922:2016; International Organization for Standardization: Geneva, Switzerland, 2016.
22. Naik, N. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In Proceedings of the 2017 IEEE International Systems Engineering Symposium (ISSE), Vienna, Austria, 11–13 October 2017. doi:10.1109/syseng.2017.8088251.

23. Provoost, M.; Weyns, D. DingNet: A self-adaptive internet-of-things exemplar. In Proceedings of the 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Montreal, QC, Canada, 25–31 May 2019; pp. 195–201. doi:10.1109/SEAMS.2019.00033.
24. Serugendo, G.D.M.; Foukia, N.; Hassas, S.; Karageorgos, A.; Mostéfaoui, S.K.; Rana, O.F.; Ulieru, M.; Valckenaers, P.; van Aart, C. Self-Organisation: Paradigms and Applications. In *Lecture Notes in Computer Science*; ESOA@AAMAS; Serugendo, G.D.M.; Karageorgos, A., Rana, O.F., Zambonelli, F., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2977, pp. 1–19. doi:10.1007/978-3-540-24701-2_1.
25. Wolf, T.D.; Holvoet, T. Emergence Versus Self-Organisation: Different Concepts but Promising When Combined. In *Lecture Notes in Computer Science*; ESOA@AAMAS; Brueckner, S., Serugendo, G.D.M., Karageorgos, A.,Nagpal, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3464, pp. 1–15. doi:10.1007/11494676_1.
26. Richter, U.; Mnif, M.; Branke, J.; Müller-Schloer, C.; Schmeck, H. Towards a generic observer/controller architecture for Organic Computing. In Proceedings of the 36 Jahrestagung der Gesellschaft für Informatik, Informatik für Menschen (INFORMATIK 2006), Dresden, Germany, 2–6 October 2006; Volume P-93, pp. 112–119.
27. Parunak, H.V.D.; Brueckner, S.; Matthews, R.S.; Sauter, J.A. Pheromone learning for self-organizing agents. *IEEE Trans. Syst. Man Cybern. Part A* **2005**, *35*, 316–326. doi:10.1109/TSMCA.2005.846408.
28. Diaconescu, A.; Marsh, S.; Pitt, J.; Reif, W.; Steghöfer, J. Social Concepts in Self-organising Systems (Dagstuhl Seminar 15482). *Dagstuhl Rep.* **2015**, *5*, 127–150. doi:10.4230/DagRep.5.11.127.
29. Torrent-Fontbona, F.; López, B.; Busquets, D.; Pitt, J.V. Self-organising energy demand allocation through canons of distributive justice in a microgrid. *Eng. Appl. Artif. Intell.* **2016**, *52*, 108–118. doi:10.1016/j.engappai.2016.02.010.
30. Rodríguez, A.; Gómez, J.; Diaconescu, A. Foraging-Inspired Self-Organisation for Terrain Exploration with Failure-Prone Agents. In Proceedings of the 2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems, Cambridge, MA, USA, 21–25 September 2015; pp. 121–130. doi:10.1109/SASO.2015.20.
31. Esterle, L. Centralised, Decentralised, and Self-Organised Coverage Maximisation in Smart Camera Networks. In Proceedings of the 2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Tucson, AZ, USA, 18–22 September 2017; pp. 1–10. doi:10.1109/SASO.2017.9.
32. Weyns, D.; Boucke, N.; Holvoet, T. A field-based versus a protocol-based approach for adaptive task assignment. *J. Auton. Agents Multi Agent Syst.* **2008**, *2*, 288–319. doi:10.1007/s10458-008-9037-x .
33. Pianini, D.; Casadei, R.; Viroli, M.; Natali, A. Partitioned integration and coordination via the self-organising coordination regions pattern. *Future Gener. Comput. Syst.* **2021**, *114*, 44–68. doi:10.1016/j.future.2020.07.032.
34. Bucchiarone, A.; Mongiello, M. Ten Years of Self-adaptive Systems: From Dynamic Ensembles to Collective Adaptive Systems. In *From Software Engineering to Formal Methods and Tools, and Back*; Lecture Notes in Computer Science; ter Beek, M.H., Fantechi, A., Semini, L., Eds.; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11865, pp. 19–39. doi:10.1007/978-3-030-30985-5_3.
35. Oreizy, P.; Gorlick, M.M.; Taylor, R.N.; Heimhigner, D.; Johnson, G.; Medvidovic, N.; Quilici, A.; Rosenblum, D.S.; Wolf, A.L. An architecture-based approach to self-adaptive software. *IEEE Intell. Syst.* **1999**, *14*, 54–62. doi:10.1109/5254.769885.
36. Cheng, B.H.C.; de Lemos, R.; Giese, H.; Inverardi, P.; Magee, J.; Andersson, J.; Becker, B.; Bencomo, N.; Brun, Y.; Cukic, B.; et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems [Outcome of a Dagstuhl Seminar]*; Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5525, pp. 1–26. doi:10.1007/978-3-642-02161-9_1.
37. Garlan, D.; Cheng, S.; Huang, A.; Schmerl, B.R.; Steenkiste, P. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer* **2004**, *37*, 46–54. doi:10.1109/MC.2004.175.
38. Weyns, D. Software Engineering of Self-adaptive Systems. In *Handbook of Software Engineering*; Springer: Cham, Switzerland, 2019; pp. 399–443. doi:10.1007/978-3-030-00262-6_11.

39. Weyns, D. *Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective*; Wiley: Hoboken, NJ, USA, 2020; ISBN 978-1-119-57494-1.
40. Salehie, M.; Tahvildari, L. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.* **2009**, *4*, 14:1–14:42. doi:10.1145/1516533.1516538.
41. Kephart, J.O.; Chess, D.M. The Vision of Autonomic Computing. *IEEE Comput.* **2003**, *36*, 41–50. doi:10.1109/MC.2003.1160055.
42. Kramer, J.; Magee, J. Self-Managed Systems: An Architectural Challenge. In Proceedings of the Future of Software Engineering (FOSE '07), Minneapolis, MN, USA, 23–25 May 2007; pp. 259–268. doi:10.1109/FOSE.2007.19.
43. Weyns, D.; Malek, S.; Andersson, J. FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.* **2012**, *7*, 8:1–8:61. doi:10.1145/2168260.2168268.
44. Blair, G.; Bencomo, N.; France, R.B. Models@ run.time. *Computer* **2009**, *42*, 22–27. doi:10.1109/mc.2009.326.
45. Weyns, D.; Malek, S.; Andersson, J. On decentralized self-adaptation: Lessons from the trenches and challenges for the future. In Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10), Cape Town, South Africa, 3–4 May 2010; pp. 84–93. doi:10.1145/1808984.1808994.
46. Musil, A.; Musil, J.; Weyns, D.; Bures, T.; Muccini, H.; Sharaf, M. Patterns for Self-Adaptation in Cyber-Physical Systems. In *Multi-Disciplinary Engineering for Cyber-Physical Production Systems, Data Models and Software Solutions for Handling Complex Engineering Projects*; Biffl, S., Lüder, A., Gerhard, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2017; pp. 331–368. doi:10.1007/978-3-319-56345-9_13.
47. Malek, S.; Mikic-Rakic, M.; Medvidovic, N. A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. In Proceedings of the Component Deployment, Third International Working Conference (CD 2005), Grenoble, France, 28–29 November 2005; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3798, pp. 99–114. doi:10.1007/11590712_8.
48. Vromant, P.; Weyns, D.; Malek, S.; Andersson, J. On interacting control loops in self-adaptive systems. In Proceedings of the 2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), Waikiki, Honolulu, HI, USA, 23–24 May 2011; pp. 202–207. doi:10.1145/1988008.1988037.
49. Vogel, T.; Giese, H. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Trans. Auton. Adapt. Syst.* **2014**, *8*, 18:1–18:33. doi:10.1145/2555612.
50. Iftikhar, U.; Weyns, D. A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System. *Electron. Proc. Theor. Comput. Sci.* **2012**, *91*. doi:10.4204/eptcs.91.4.
51. Calinescu, R.; Gerasimou, S.; Banks, A. Self-adaptive Software with Decentralised Control Loops. In Proceedings of the 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, 11–18 April 2015; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9033, pp. 235–251. doi:10.1007/978-3-662-46675-9_16.
52. Weyns, D.; Georgeff, M. Self-Adaptation Using Multiagent Systems. *IEEE Softw.* **2010**, *27*, 86–91. doi:10.1109/MS.2010.18.
53. Haesevoets, R.; Weyns, D.; Holvoet, T. Architecture-centric support for adaptive service collaborations. *ACM Trans. Softw. Eng. Methodol.* **2010**, *23*, 2:1–2:40. doi:10.1145/2559937.
54. Caporuscio, M.; Grassi, V.; Marzolla, M.; Mirandola, R. GoPrime: A Fully Decentralized Middleware for Utility-Aware Service Assembly. *IEEE Trans. Softw. Eng.* **2016**, *42*, 136–152. doi:10.1109/TSE.2015.2476797.
55. Heylighen, F. Stigmergy as a universal coordination mechanism I: Definition and components. *Cogn. Syst. Res.* **2016**, *38*, 4–13. doi:10.1016/j.cogsys.2015.12.002.
56. de Lemos, R.; Giese, H.; Müller, H.A.; Shaw, M.; Andersson, J.; Litoiu, M.; Schmerl, B.; Tamura, G.; Villegas, N.M.; Vogel, T.; et al. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II*; de Lemos, R., Giese, H., Müller, H.A., Shaw, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; Volume 7475, pp. 1–32. doi:10.1007/978-3-642-35813-5_1.
57. Bicocchi, N.; Mamei, M.; Zambonelli, F. Self-organizing virtual macro sensors. *ACM Trans. Auton. Adapt. Syst.* **2012**, *7*, 2:1–2:28. doi:10.1145/2168260.2168262.
58. Beal, J.; Bachrach, J. Infrastructure for Engineered Emergence on Sensor/Actuator Networks. *IEEE Intell. Syst.* **2006**, *21*, 10–19. doi:10.1109/MIS.2006.29.

59. Schmeck, H. Organic Computing—A New Vision for Distributed Embedded Systems. In Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), Seattle, WA, USA, 18–20 May 2005; pp. 201–203. doi:10.1109/ISORC.2005.42.
60. Ballouli, R.E.; Bensalem, S.; Bozga, M.; Sifakis, J. Four Exercises in Programming Dynamic Reconfigurable Systems: Methodology and Solution in DR-BIP. In *Leveraging Applications of Formal Methods, Verification and Validation, Distributed Systems—8th International Symposium, ISoLA 2018, Limassol, Cyprus, 5–9 November 2018, Proceedings, Part III*; Margaria, T., Steffen, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2018; Volume 11246, pp. 304–320. doi:10.1007/978-3-030-03424-5_20.
61. Nicola, R.D.; Maggi, A.; Sifakis, J. The DReAM framework for dynamic reconfigurable architecture modelling: Theory and applications. *Int. J. Softw. Tools Technol. Transf.* **2020**, *22*, 437–455. doi:10.1007/s10009-020-00555-2.
62. Aceto, L.; Fokkink, W.; Verhoef, C. CHAPTER 3—Structural Operational Semantics. In *Handbook of Process Algebra*; Bergstra, J., Ponse, A., Smolka, S., Eds.; Elsevier Science: Amsterdam, The Netherlands, 2001; pp. 197–292. doi:10.1016/B978-044482830-9/50021-7.
63. Milner, R.; Parrow, J.; Walker, D. A Calculus of Mobile Processes, I. *Inf. Comput.* **1992**, *100*, 1–40. doi:10.1016/0890-5401(92)90008-4.
64. Audrito, G.; Viroli, M.; Damiani, F.; Pianini, D.; Beal, J. A Higher-Order Calculus of Computational Fields. *ACM Trans. Comput. Log.* **2019**, *20*, 1–55. doi:10.1145/3285956.
65. Fischer, A.; Botero, J.F.; Beck, M.T.; de Meer, H.; Hesselbach, X. Virtual Network Embedding: A Survey. *IEEE Commun. Surv. Tutor.* **2013**, *15*, 1888–1906. doi:10.1109/SURV.2013.013013.00155.
66. Viroli, M.; Beal, J.; Damiani, F.; Audrito, G.; Casadei, R.; Pianini, D. From distributed coordination to field calculus and aggregate computing. *J. Log. Algebr. Methods Program.* **2019**, *109*, 100486. doi:10.1016/j.jlamp.2019.100486.
67. Beal, J.; Dulman, S.; Usbeck, K.; Viroli, M.; Correll, N. Organizing the Aggregate: Languages for Spatial Computing. *arXiv* **2012**, arXiv:1202.5509.
68. Saelens, M.; Kinoo, Y.; Weyns, D. HeyCitI: Healthy Cycling in a City using Self-Adaptive Internet-of-Things. In Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS), Washington, DC, USA, USA, 17–21 August 2020. doi:10.1109/ACSOS-C51401.2020.00061.
69. Blenn, N.; Kuipers, F.A. LoRaWAN in the Wild: Measurements from The Things Network. *arXiv* **2017**, arXiv:1706.03086.
70. Pianini, D.; Elzanaty, A.; Giorgetti, A.; Chiani, M. Emerging Distributed Programming Paradigm for Cyber-Physical Systems Over LoRaWANs. In Proceedings of the 2018 IEEE Globecom Workshops (GC Wkshps), Abu Dhabi, UAE, 9–13 December 2018. doi:10.1109/glocomw.2018.8644518.
71. Xiao, Q.; Ma, Z.; Li, S.; Liu, Y. The Impact of Winter Heating on Air Pollution in China. *PLoS ONE* **2015**, *10*, e0117311. doi:10.1371/journal.pone.0117311.
72. Sonmez, C.; Ozgovde, A.; Ersoy, C. EdgeCloudSim: An environment for performance evaluation of edge computing systems. *Trans. Emerg. Telecommun. Technol.* **2018**, *29*. doi:10.1002/ett.3493.
73. van den Elshout, S.; Léger, K.; Nussio, F. Comparing urban air quality in Europe in real time. *Environ. Int.* **2008**, *34*, 720–726. doi:10.1016/j.envint.2007.12.011.
74. Olszowski, T. Changes in PM10 concentration due to large-scale rainfall. *Arab. J. Geosci.* **2016**, *9*. doi:10.1007/s12517-015-2163-2.
75. Grimmett, R. *Getting Started with Raspberry Pi Zero*; Packt Publishing Ltd., Birmingham, UK, 2016.
76. Hoyer, S.; Hamman, J. xarray: N-D labeled arrays and datasets in Python. *J. Open Res. Softw.* **2017**, *5*, 10. doi:10.5334/jors.148.
77. Hunter, J.D. Matplotlib: A 2D Graphics Environment. *Comput. Sci. Eng.* **2007**, *9*, 90–95. doi:10.1109/MCSE.2007.55.
78. Pianini, D.; Ciatto, G.; Casadei, R.; Mariani, S.; Viroli, M.; Omicini, A. Transparent Protection of Aggregate Computations from Byzantine Behaviours via Blockchain. In Proceedings of the 4th EAI International Conference on Smart Objects and Technologies for Social Good—Goodtechs, Bologna, Italy, 28–30 November 2018. doi:10.1145/3284869.3284870.
79. Casadei, R.; Aldini, A.; Viroli, M. Towards attack-resistant Aggregate Computing using trust mechanisms. *Sci. Comput. Program.* **2018**, *167*, 114–137. doi:10.1016/j.scico.2018.07.006.

80. Adelantado, F.; Vilajosana, X.; Tuset-Peiró, P.; Martínez, B.; Melià-Seguí, J.; Watteyne, T. Understanding the Limits of LoRaWAN. *IEEE Commun. Mag.* **2017**, *55*, 34–40. doi:10.1109/MCOM.2017.1600613.