Dynamic Decentralization Domains for the Internet of Things

Journal:	*IEEE Internet Computing*
Manuscript ID	ICSI-2022-04-0040.R2
Manuscript Type:	SI: Nov/Dec 2022 Decentralized Systems
Date Submitted by the Author:	n/a
Complete List of Authors:	Aguzzi, Gianluca; Alma Mater Studiorum Università di Bologna, Department of Computer Science and Engineering Casadei, Roberto; Alma Mater Studiorum Università di Bologna, Department of Computer Science and Engineering Pianini, Danilo; Alma Mater Studiorum Università di Bologna, Department of Computer Science and Engineering Viroli, Mirko; Alma Mater Studiorum Università di Bologna, Department of Computer Science and Engineering
Keywords:	K.6.4.a Centralization/decentralization < K.6.4 System Management < K.6 Management of Computing and Information Systems < K Computing Milieux, D.1.8 Distributed programming < D.1 Programming Techniques < D Software/Software Engineering, J.9.d Pervasive computing < J.9 Mobile Applications < J Computer Applications, D.2.11.b Domain-specific architectures < D.2.11 Software Architectures < D.2 Software Engineering < D Software/Software Engineer, I.2.1.n Space < I.2.1 Applications and Expert Knowledge-Intensive Systems < I.2 Artificial Intelligence < I Computing Methodolog, I.2.11.d Multiagent systems < I.2.11 Distributed Artificial Intelligence < I.2 Artificial Intelligence < I Computing Methodologi, I.2.11.a Coherence and coordination < I.2.11 Distributed Artificial Intelligence < I.2 Artificial Intelligence < I Computing Met

Department: Head
Editor: Name, xxxx@email

Dynamic Decentralization Domains for the Internet of Things

Gianluca Aguzzi
Alma Mater Studiorum–Università di Bologna

Roberto Casadei
Alma Mater Studiorum–Università di Bologna

Danilo Pianini
Alma Mater Studiorum–Università di Bologna

Mirko Viroli
Alma Mater Studiorum–Università di Bologna

***Abstract*—The Internet of Things and edge computing are fostering a future of ecosystems hosting complex decentralized computations, deeply integrated with our very dynamic environments. Digitalized buildings, communities of people, and cities will be the next-generation "hardware and platform", counting myriads of interconnected devices, on top of which intrinsically-distributed computational processes will run and self-organize. They will spontaneously spawn, diffuse to pertinent logical/physical regions, cooperate and compete, opportunistically summon required resources, collect and analyze data, compute results, trigger distributed actions, and eventually decay.**
How would a programming model for such ecosystems look like? Based on research findings on self-adaptive/self-organizing systems, this paper proposes design abstractions based on "dynamic decentralization domains": regions of space opportunistically formed to support situated recognition and action. We embody the approach into a Scala application program interface (API) enacting distributed execution and show its applicability in a case study of environmental monitoring.

■ **E**dge computing and related scenarios like the Internet of Things (IoT) and cyber-physical systems (CPSs) promote a vision of distributed computational systems deeply integrated with humans and environments. The complexity and volume in terms of devices, communications, failures, and change, are pushing the adoption of paradigms that can adequately address both functional and non-functional concerns:

- *decentralization* for scalability and delegation;
- *autonomic computing* and *self-organization* [1] for operational effectiveness and adaptation;
- *in-network processing* for latency reduction and infrastructural autonomy; and
- *collective computing* [2] for coordination and collaboration.

We envision environments (bodies, rooms, buildings, communities, cities) populated by myriads of devices whose ensemble will be abstracted as a single programmable CPS. These entities may or may not be coordinated in a centralized fashion, namely, we cannot assume a central coordinator (e.g., the cloud) exists when designing the system. They form the platform upon which several concurrent *distributed computational processes (DCPes)* would run, carrying on transient activities by self-organized continuous computation and communication. The goal of a DCP is to identify dynamic regions of the computational environment (regions of "space") where situations of interest occur, monitor their evolution, and reactively trigger distributed actions to signal events, remedy problems, or control the phenomenon. Hence, DCPes are generated to satisfy a request, handle an event, or execute a collective task; they opportunistically spread (resp. shrinks) to gather (resp. release) resources/workers or cover (resp. uncover) regions of interest; they may perform distributed sensing and actuation; eventually, they may vanish once the activity is done.

In this paper, we address the problem of capturing the right abstractions for modeling DCPes, abstracting from the specific communication technologies, and propose the concepts of *concurrent collective tasks* and *decentralization domains*, which can be exploited in combination to provide distributed situated recognition and action.

In Section 1, we motivate the proposal, identify three essential requirements, and briefly summarize the state of the art. In Section 2, we detail the technical solution, providing a declarative API. In Section 3, we functionally evaluate the proposal through simulation in an environmental monitoring case study. Our finding and perspective, detailed in Section 4, is that the programming model provides a high-level yet expressive framework for DCPes, with fine-grained control over decentralization.

1. Background and Motivation

Declarative abstractions for complex decentralized IoT systems

Modern computing ecosystems such as IoT ones are increasingly complex and resourceful, providing opportunities turning into functional and non-functional goals. The recurrent approach in computer/software engineering to harness complexity is to adopt *levels* of abstractions and mechanisms encapsulating coherent sets of problems and solutions.

This work focuses on *situated* distributed systems that need to monitor and act on a dynamically changing environment, and where a central coordinator may not be available. Typical examples include crowd tracking and steering, environmental (landslides, floodings, fires) monitoring and response, resource allocation in open systems, and coordination of robot swarms. Our *target system model* is a *network* of computing and communicating *devices*. Every device may interact with a limited subset of other devices (its *neighbors*).

We target the idea of programming the overall behavior of such systems by expressing a high-level goal—e.g., monitoring the safety of an environment by integrating recent data from static and mobile sensors, then computing local suggestions for risk-mitigating actions. However, we would not *fully* specify *how* activities should concretely be carried out, as long as these decisions do not affect the intended result. More specifically, we intend to declaratively express *what* is to be achieved, letting lower-level components deal with issues like handling dynamicity (e.g., due to mobility or openness), failure, heterogeneity, etc.

As an analogy, consider database management systems: queries express what data has to be retrieved, and the query optimizer determines an efficient query plan satisfying the request. Wireless sensor network (WSN) macroprogramming approaches [3] are another example where declarative queries get mapped to data processing and transfer operations carried out across sensor nodes and base stations. We aim to apply the same principle to self-organizing systems, primarily to realize decentralized situation recognition and action.

Decentralized situation recognition and action: a case study

A self-organizing IoT system should ideally determine autonomously *what* has to be done, *when*, *where*, by *whom*, and *how*. The critical problem is setting up a *decentralized process for adaptive situation recognition and situated action*. The system should organize to monitor the environment for situations requiring intervention; then, the intervention should pursue the desired state of affairs. Also, we cannot assume the existence of a centralized coordinator such as the cloud, which is usually relied upon in classic approaches.

As an example and case study throughout the paper, consider a large-scale flood warning system, which we call FLOODWATCH, fully developed (in simulation) in Section 3. We want to monitor the rain intensity to pre-alert the public safety organizations close to areas at a *risk* of floods. The tracked phenomenon is spatially and temporally hard to predict with fine-enough grain (data from the NOAA[1] has, at best, zip-code granularity): at a single-city level, we could perform better by promptly reacting to specialized sensors readings. However, the information provided by individual sensors is too fragile, as the risk depends on the rain intensity in surroundings and not just on the specific spot (e.g., coastal zones with a steep elevation profiles could suffer floods even with light rain, if the close-by higher-altitude zone is being hit hard). Pre-defining areas (using pre-existing altimetric and structural knowledge) helps, but this strategy misses out on essential information: how the underlying phenomenon is behaving. Indeed, areas should be formed ad-hoc considering the city structure and rain distribution, and leveraged to perform on-the-fly situation recognition and response.

This approach is practical whenever there are phenomena with non-strictly-local effects, irregularly shaped in space, and/or hard-to-predict at a fine grain.

[1]https://www.noaa.gov/

Requirements and abstractions

Given the high-level vision and goals discussed in the previous sections, and with the help of FLOODWATCH, we delineate some *needs* together with *abstractions* and corresponding *requirements*, for a programming model aimed at decentralized situation recognition and action.

R1. Concurrent collective task execution.
In FLOODWATCH, there is the need to coordinate a system that spans large geographical areas, hence leveraging DCPes for sensing, computation, and actuation at a collective level.

Most complex systems involve several activities running concurrently. Furthermore, these activities could be collective, i.e., involve a collaboration of multiple agents with partial perception of the environment. We call these *concurrent collective tasks (CCTs)*, which express activities that may *overlap* in the system (a device may partake in multiple CCTs simultaneously). Notice that CCTs may have a limited and dynamic domain: a subset of devices which may change over time.

R2. Flexible and adaptive decentralization.
FLOODWATCH is centered on organizing distributed sensing and actuation according to both the environment structure and the current rainfall. Generally speaking, strategies that are too fine-grained or too coarse-grained tend to be suboptimal: in the former case, non-local information is not considered, possibly resulting in a lack of coherence and global inefficiency; in the latter case, the system may fail to adequately recognize specific contexts that should be handled ad-hoc. In FLOODWATCH, warnings should be delivered in the surroundings of risky areas, but not too broadly.

Many systems, indeed [4], often need abstractions capturing an “adaptive” *spatial divide-and-conquer* principle through which a problem in space is split into parts (or regions) that opportunistically adapt according to the context. We call each region a *decentralization domain (DD)* since it represents a non-overlapping bounded subsystem of a CCT. Multiple DDs can also *compete* to gather resources exclusively within the domain defined by a CCT (at whose level cross-domain interaction could happen, instead).

R3. Feedback-regulated activity within decentralized domains.
In FLOODWATCH, each region should sense the water level and altitude, process data, and decide region-local actions such as alerting. In a system for computational resource management, each region might collect resource advertisements and requests, compute assignments, and publish assignments while also monitoring and handling the activity progress.

In general, DDs are expected to autonomously carry out distributed sensing activities, followed by processing and decision-making, which may trigger actions affecting the environment or spawning new CCTs (e.g., to connect with other services).

Summary of requirements.
The rationale of the above requirements is to promote abstractions supporting concurrent, system-spanning, and possibly overlapping activities (R1), dynamic creation and maintenance of non-overlapping regions (R2), and internal loops of regional situation recognition and action (R3). Figure 1 summarizes these ideas.

Related problems and state-of-the-art approaches

The stance by which the behavior of a whole system is expressed as a single program is shared by paradigms like macro-programming [3], field-based computing [5], spatial computing [6], and aggregate computing [7]. The programming model that we provide is best understood as a higher-level wrapper on aggregate and field-based computations.

These approaches provide means for expressing *collective* tasks and processes [8], [2] performed by *ensembles* [9] of devices. From [2], we reuse the idea of *aggregate processes*, i.e., computations that spring out, spread, shrink, and vanish to express transient collective operations. CCTs can be thought of as a generalization of aggregate processes, and also differ from *collective-based tasks* [8] which are based on orchestration.

The programming model covered in this article is also related to *self-organizing coordination regions (SCR)* [4], a design pattern promoting divide-and-conquer through dynamic feedback-regulated regional partitioning process. More generally, the importance of structured communities in multi-agent systems is witnessed by the sheer number of *organizational paradigms* [10]. The abstractions presented in this paper sit at a higher level and may be seen as implementatively exploiting SCR and organizational patterns to solve the problem of decentralized situation recognition and action. Overall, the contribution of this work lies primarily in the *combination* of the discussed abstractions into a simple but effective API.

2. Dynamic Decentralization Domains in Practice

Programming Model

We assume a programming model where abstractions drive an entire system of devices. So, we aim at a *macro*-programming model [11], where a single program defines the behavior of the whole system by a global perspective. In particular, we assume distributed execution protocol consisting of *repeated* rounds of *sensing, processing, and neighbor-based communication*, and let the program specify what data has to be sensed and exchanged and what processing has to be applied to it. We also want the API to be *declarative*, characterizing the rules promoting the behavior of the abstractions identified in the previous section, and abstracting from low-level details like the details of scheduling, networking, or deployment. In particular, the program may be deployed and evaluated on all the devices constituting the system, or may be computed on behalf of them by a distinct managing system—following the approach of [12].

From Requirements to an API

From the previous discussion, we further refine the requirements and extrapolate the design elements of an API supporting the decentralized computation we need:

- concerning CCTs (cf. R1)
  - we use a CCT to model a collective sensing task partitioned into multiple *sensing domains* (i.e. DDs), where each sensing domain has a *center* and an *extension* in space;
  - both the extension in space and the center can change dynamically to improve the way the underlying phenomenon is being tracked, through selection of an

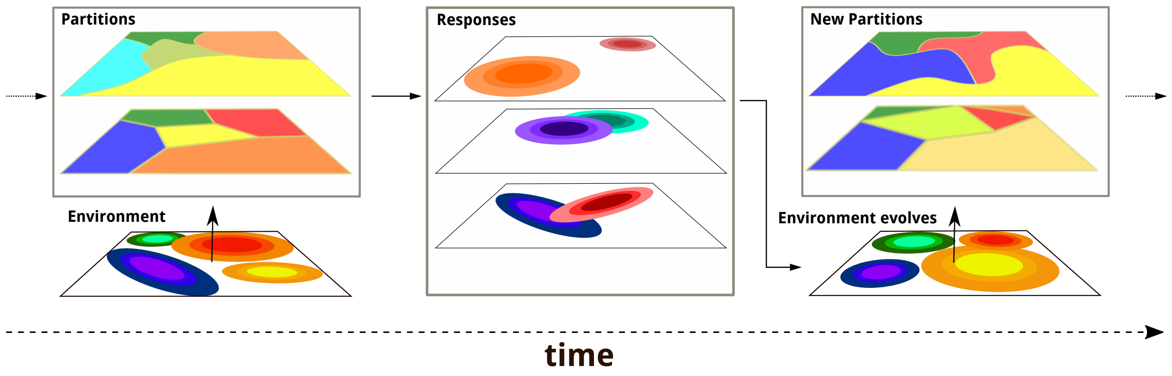

Figure 1: Overview of the proposed approach. The projected squares represent environments, with colors denoting environmental phenomena. Time flows left to right. The proposed system tracks spatial phenomena by partitioning the space in non-overlapping regions that agree on a measure, which is then leveraged to enact (multiple) spatially-bound responses that can overlap or compete. Contextual (induced or natural) changes are tracked by evolving (reshaping, deleting, or creating) the partitions.

appropriate leading node, definition of a metric (which can be other than the spatial distance), and definition of a granularity.

- concerning partitioning into DDs (cf. R2) and activity within a DD (cf. R3)
  - sensing domains for a single measure must not overlap, to avoid duplicate sampling and undesired interference (overlapping can be achieved through multiple CCTs, or by using a mixed custom metric);
  - inside a single sensing domain, a strategy is defined to collect the sensor readings;
  - the decentralized sensing will output the collectively-sensed result and the identifier of the device closer to the area center;
  - the set of actions/actuations to perform may vary depending on the overall sensing results, could require a collective plan for coordination, and may require fine-grained information about all the results of the sensing phase.

To the best of our knowledge, no completely decentralized API/framework exists in the literature that directly satisfies the aforementioned requirements (although, of course, it can be implemented leveraging existing frameworks). Thus, we designed a Scala API, presented in Figure 2a, which serves two roles: *(i)* to reify the sought abstractions, and hence as a specification tool for dynamic decentralization domains; and *(ii)* as a basis for a prototypical implementation on top of the ScaFi framework [2], [13], which will be presented and used in the experiments in next section. Specifically, class `DistributedSensing` denotes DDs; types `Perception`, `SituatedRecognition`, and `Action` model sensing, reasoning, and acting operations, respectively; and `decentralisedRecognitionAndResponse` encapsulates the logic that creates multiple CCTs and manages their dynamic partitioning into DDs.

Consider the FloodWatch case study introduced in Section 1 as a reference scenario. We assume that several pluviometers, deployed in the city, can communicate with each other. We want to monitor the progression of a storm hitting the city, adjusting the granularity at runtime: large areas with similar rain intensity should get clustered together; if, instead, the precipitation is spotty, each spot should form a region. In other words, we want to leverage the clustering of similarly affected areas to achieve a better global tracking of the underlying phenomenon, understand its spatial structure, and potentially exploit the information for better counteraction.

We assume that lower parts of the city are at a higher risk in case of floods. We assume that the rain gauges have a GPS sensor supporting altimetry measurement (we would like to consider this information when responding to a potential emergency). Finally, we want to consider the altimetry of an entire zone and not of a single

```scala
/* Configuration of a distributed sensing task */
class DistributedSensing[Leadership,Distance,Data](
  perceptionCenter: () => Leadership,
  localValue: () => Data,
  metric: Data => Distance,
  accumulate: UnivariateStatistics[Data],
  limit: Distance
){
  def compute(): Data = /* API implementation */
}

type Perception[Data] =// Collective sensing result
  Map[DistributedSensing[?, ?, Data], Data]
type Action = () => Unit // Response action type

/* Situation recognition: perception to action */
type SituatedRecognition[Data] =
  Perception[Data] => Set[Action]

// Actual high level API
def decentralizedRecognitionAndResponse[Data](
    sensing: Set[DistributedSensing[?, ?, Data]],
    situatedRecognition: SituatedRecognition[Data]
): Unit = {
  /* CCT creation and Action execution */
}
```

(a) Scala API for decentralized situation recognition

```scala
// FloodWatch program
type FloodWatchSensing =
  DistributedSensing[ID, Double, Double]
val altimetry: FloodWatchSensing =
  new DistributedSensing(/**/)
val rainIntensity: FloodWatchSensing =
  new DistributedSensing(/**/)
val sensing = Set(altimetry, rainIntensity)

def propagateAlarm(): Action = ???
def callForHelp(): Action = ???

val response: SituatedRecognition[Double] =
  situation => {
    /* create actions related to alarms */
    val alarm: Set[Action] = ???
    /* call fire station following alarms */
    val call: Set[Action] = ???
    alarm ++ call
}

// Actual API usage
decentralizedRecognitionAndResponse(
  sensing,
  response
)
```

(b) Example use of the API for the case study

Figure 2: Scala implementation of the proposed API, showing the abstractions (a) and their exemplary use (b). `DistributedSensing` represents the configuration of the collective value-reading operation, that selects a leading node, expands an area of influence, and produces an area-wide result; `Action` represents a collective task enacted in response to a distributed perception; `Perception` links each distributed sensing process to the corresponding computed value (i.e., the result of the collective sensing process); `SituatedRecognition` maps collective perceptions to actual actions; `decentralizedRecognitionAndResponse` is the entry point.

point, and to react promptly if any rain gauge is moved to a different location: we thus use the same technique for both rain intensity and altimetry.

The application goal goes beyond sensing: when the rain in low-altitude areas is so heavy that it might cause floods, we want to:

1) propagate an alert signal to the surroundings of the area at risk, to be perceived, e.g., by smart vehicles transiting by; and
2) pre-alert the closest fire station or civil protection post to be prepared in case of actual issues.

The application logic, leveraging our API, is shown in Figure 2b. As detailed in [13], [2], this specification is also the "script" that each device executes repeatedly in asynchronous *sense–compute–interact* rounds, progressively building the intended global behavior.

3. Evaluation

In this section, we consider the FLOODWATCH case study, and show that our API can successfully be used in a challenging scenario to program a system behavior that responds as expected to the underlying environmental phenomena.

Experimental setup

We exercise the API in a challenging and realistic scenario, using open data of Toronto[2], featuring 50 water gauges samples taken in 2021. To stress-test our proposed approach with a denser network of devices, we added 300 simulated gauges, randomly positioned, whose data is interpolated from the values of the surrounding real devices. We selected the rain event that occurred on 2021-09-07, the heaviest in the available data. We used data from OpenStreetMap[3] to position 24 fire stations.

We implemented the proposed API in the SCAFI aggregate programming toolkit [13] and

[2]https://bit.ly/3QciJ9i
[3]using Overpass API https://overpass-turbo.eu/

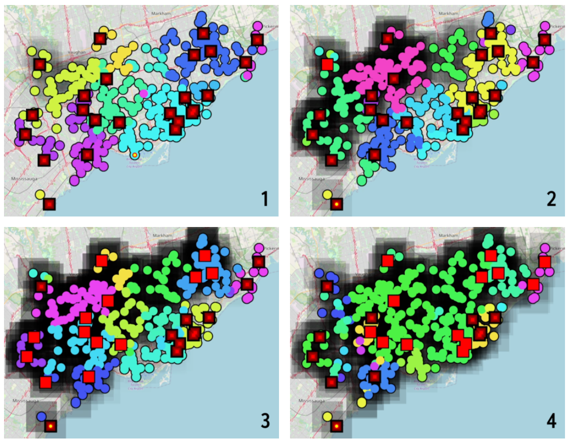

Figure 3: Subsequent simulation snapshots (left-to-right, top-to-bottom) of FLOODWATCH. Darker shadows indicate heavier rain. Black squares with a small red dot are unalerted fire stations, when at least an alert reaches them, their dot changes to a large red square. Circles represent gauges; their colors map the DDs they are subject to when measuring rainfall intensity. A video of a complete simulation run is available at https://bit.ly/3zysMOV.

simulated the scenario using the Alchemist simulator [14]. In the experiment, devices compute their programs unsynchronized at a frequency of 1Hz. We define a simple metric for the actual risk of a location as the quotient of the local rain intensity on the local altitude (namely, the rainier and the lower the position, the higher the risk); we run an oracle measuring it with a fine grain across the city at each instant. As performance measure, we count how many alerts get generated and how many stations they reach. Additional gauges position and device timing drift are randomized. We ran 64 repetitions of the simulation and considered the mean results. The experiment is available and reproducible; it has been released, open-sourced[4], and permanently archived [15]. Figure 3 depicts the scenario as simulated in Alchemist.

Results and discussion

Figure 3 shows that, when conditions change, DDs adapt by changing their shape and extension to track the underlying phenomenon coherently; in response to heavy rain, close-by stations get appropriately alerted. The system macroscopically tracks the underlying phenomena: more operators get alerted when (Figure 4) and where (Figure 5) there are peaks in the signal. However, even in response to similarly high peaks the system may decide to allocate less or more resources to manage them: differences are primarily due to the system detecting different base risks (due to the altitude) or the event being strictly local.

We now give some remarks to better contextualize the contribution. Regarding applicability and generality, we observe that CCTs and partitioning into DDs enable addressing several kinds of applications in domains like computing ecosystems, WSNs, IoT and smart city, and multi-robot/multi-agent systems—cf. the surveys in [4], [5]. Details on quantitative cost/performance considerations on this kind of paradigm, can be found in [4], [2]: the focus of this article is on programming abstractions for decentralized systems, following a language-based software engineering approach [16].

4. Conclusion and Outlook

Mechanisms based on decentralization and self-organization are intensely researched and expected to play crucial roles in next-generation applications involving cyber-physical collectives. In this work, to turn decentralized activities into actionable notions, we propose a high-level programming model for situation recognition and action that originally integrates recent developments in collective adaptive computing. The idea is to expose a declarative API featuring *(i)* concurrent collective tasks which overlap in space and *(ii)* non-overlapping decentralization domains with inner information flows-based feedback loops. We implement the API in Scala by mapping CCTs and DDs to SCAFI aggregate computations, then show the approach's effectiveness through a case study in flood monitoring and control. Results show that programs expressed declaratively through the API yield DDs that can adapt to properly handle distributed monitoring and action.

This work focused on designing and programming decentralized systems. We believe that this level of control is instrumental for properly structuring collective adaptive behavior to steer desired emergents. Yet, contributions on patterns and pro-

[4]https://bit.ly/3vF09P6

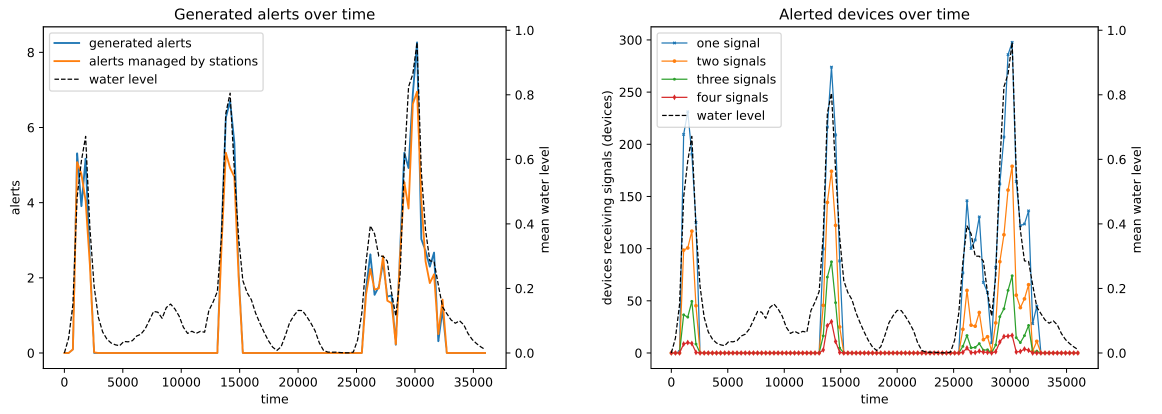

Figure 4: Simulation results, showing, with time, the average rainfall intensity (dotted black line), the number of operator stations receiving alerts (left) and the breakdown by number of alerts received per station (right).

gramming abstractions for this class of systems are still quite fragmented. Further, their integration with automatic design approaches may be a fertile path for future research.

Acknowledgment

This work was supported by the Italian PRIN projects "Fluidware" (2017KRC7KT) and "CommonWears" (2020HCWWLP), and the EU/MUR FSE PON-R&I 2014-2020.

Aguzzi Gianluca is a PhD student at the Department of Computer Science and Engineering of the University of Bologna (Italy), from which he received his master degree in computer science and software engineering. His research interests include software engineering, pervasive systems, and multi-agent reinforcement learning.

Roberto Casadei is a post-doc researcher at the University of Bologna (Italy). He has a PhD in Computer Science and Engineering from the same university, with a thesis awarded by IEEE TCSC. His research interests revolve around software engineering and distributed artificial intelligence. He has 40+ publications in international journals and conferences on topics including collective intelligence, self-* systems, and IoT/CPS.

Danilo Pianini is a post-doctoral researcher at the Department of Computer Science and Engineering of the University of Bologna (Italy). He holds a PhD in Computer Science and Engineering, and his main research interests include simulation, (self-organizing) coordination, aggregate computing, pervasive systems, software engineering, agile techniques, and DevOps. He published over 60 articles in international journals and conferences on those subjects.

Mirko Viroli is Full Professor in Computer Engineering at the University of Bologna (Italy), from which he received his PhD. He is an expert in foundations of computer science and programming, object-oriented programming, advanced software development, software engineering and self-adaptive/self-organizing pervasive computing systems. He is author of more than 300 papers, of which more than 70 on international journals.

REFERENCES

1. J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003. [Online]. Available: https://doi.org/10.1109/MC.2003.1160055
2. R. Casadei, M. Viroli, G. Audrito, D. Pianini, and F. Damiani, "Engineering collective intelligence at the edge with aggregate processes," *Eng. Appl. Artif. Intell.*, vol. 97, p. 104081, 2021. [Online]. Available: https://doi.org/10.1016/j.engappai.2020.104081
3. L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, 2011. [Online]. Available: https://doi.org/10.1145/1922649.1922656
4. D. Pianini, R. Casadei, M. Viroli, and A. Natali, "Partitioned integration and coordination via the self-organising coordination regions pattern," *Future Gener.*

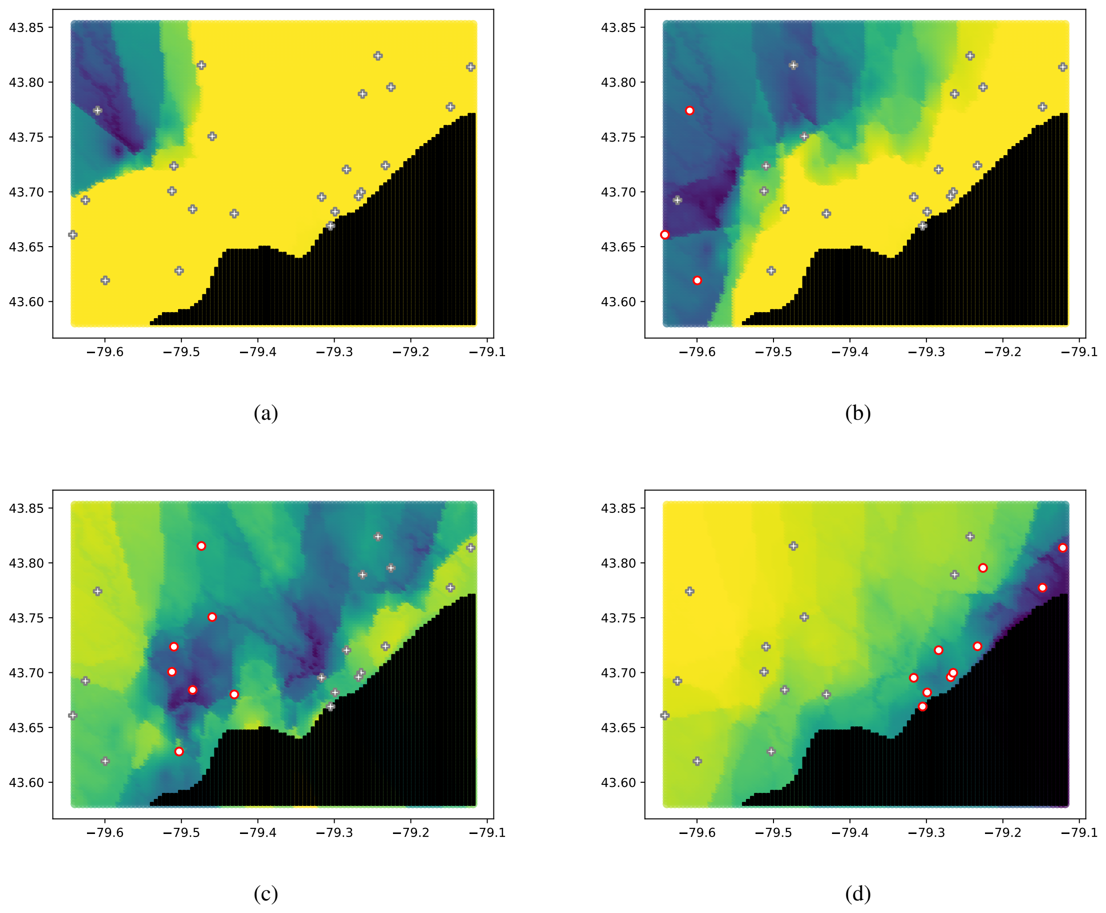

Figure 5: Ability to track the risk spatially. Charts show risk (the darker, the higher) as estimated in realtime by an oracle using altitude and rainfall intensity. Stations are depicted with a red circle, and those alerted are filled in white. Solid black areas are non-land. Alerted stations are indeed those closest to the zones of highest risk.

Comput. Syst., vol. 114, pp. 44–68, 2021. [Online]. Available: https://doi.org/10.1016/j.future.2020.07.032

5. M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, "From distributed coordination to field calculus and aggregate computing," *J. Log. Algebraic Methods Program.*, vol. 109, 2019. [Online]. Available: https://doi.org/10.1016/j.jlamp.2019.100486
6. M. Duckham, *Decentralized Spatial Computing - Foundations of Geosensor Networks*. Springer, 2013. [Online]. Available: https://doi.org/10.1007/978-3-642-30853-6
7. J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the internet of things," *Computer*, vol. 48, no. 9, pp. 22–30, 2015. [Online]. Available: https://doi.org/10.1109/MC.2015.261
8. O. Scekic, T. Schiavinotto, S. Videnov, M. Rovatsos, H. L. Truong, D. Miorandi, and S. Dustdar, "A programming model for hybrid collaborative adaptive systems," *IEEE Trans. Emerg. Top. Comput.*, vol. 8, no. 1, pp. 6–19, 2020. [Online]. Available: https://doi.org/10.1109/TETC.2017.2702578
9. T. Bures, F. Plasil, M. Kit, P. Tuma, and N. Hoch, "Software abstractions for component interaction in the internet of things," *Computer*, vol. 49, no. 12, pp. 50–59, 2016. [Online]. Available: https://doi.org/10.1109/MC.2016.377
10. B. Horling and V. R. Lesser, "A survey of multi-agent organizational paradigms," *Knowl. Eng. Rev.*, vol. 19, no. 4, pp. 281–316, 2004. [Online]. Available: https://doi.org/10.1017/S0269888905000317
11. R. Casadei, "Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour

modelling," *CoRR*, vol. abs/2201.03473, 2022. [Online]. Available: https://arxiv.org/abs/2201.03473

12. R. Casadei, D. Pianini, A. Placuzzi, M. Viroli, and D. Weyns, "Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment," *Future Internet*, vol. 12, no. 11, p. 203, 2020. [Online]. Available: https://doi.org/10.3390/fi12110203
13. R. Casadei, M. Viroli, G. Audrito, and F. Damiani, "Fscafi : A core calculus for collective adaptive systems programming," in *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Proceedings, Part II*, ser. LNCS, vol. 12477. Springer, 2020, pp. 344–360. [Online]. Available: https://doi.org/10.1007/978-3-030-61470-6_21
14. D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with ALCHEMIST," *Journal of Simulation*, vol. 7, no. 3, pp. 202–215, Aug. 2013. [Online]. Available: https://doi.org/10.1057/jos.2012.27
15. G. Aguzzi and D. Pianini, "cric96/experiment-2022-ieee-decentralised-system: 1.0.1," 2022. [Online]. Available: https://zenodo.org/record/6477039
16. G. Gupta, "Language-based software engineering," *Sci. Comput. Program.*, vol. 97, pp. 37–40, 2015. [Online]. Available: https://doi.org/10.1016/j.scico.2014.02.010