Dynamic Decentralization Domains for the Internet of Things

Journal:	*IEEE Internet Computing*
Manuscript ID	Draft
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 decade.**

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) to enact distributed execution on a network of devices and show its applicability in a case study of environmental monitoring.

Edge 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. Such systems will be characterized by countless devices or agents requesting and/or supporting resilient execution of

applications and services. The complexity and volume in terms of devices, communications, failure, and change, are pushing the adoption of paradigms that can embrace these challenges to 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 inherently capturing 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 CPSs. They form hardware and platforms upon which several concurrent *distributed computational processes (DCPes)* would run, carrying on transient activities by self-organized continuous computation and communication steps. The goal of a DCP is to identify dynamic regions of the computational environment (regions of "space") where situations of interest occur, then to monitor their evolution, and reactively trigger distributed actions to signal events, remedy problems, or control the phenomenon at hand. Hence, in its lifecycle, a DCP is generated to satisfy a request, handle an event, or execute a collective task; it opportunistically spreads (resp. shrinks) across the system to gather (resp. release) resources and workers or cover (resp. uncover) logical or physical regions of interest; it may perform distributed sensing and actuation activities; eventually, it 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 underlying 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 related work and the state of the art. In Section 2, we detail the technical solution and provide a declarative API in the Scala programming language. 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 in systems.

1. Background and Motivation

Declarative abstractions for complex decentralized IoT systems

Modern information and communication technology (ICT) systems such as IoT ones are increasingly complex and resourceful, providing opportunities that can be reified as ambitious application goals, both functional and non-functional. The recurrent approach in computer and software engineering to harness complexity is to adopt *levels* of abstractions and mechanisms encapsulating coherent sets of problems and solutions.

Our *target system model* is a logical or physical *network* of computing and communicating *devices*. A device, at some time, may interact with a limited subset of other devices known as its *neighbors*. We are mainly interested in tasks and activities that last in time (transient or ongoing), namely, that require *multiple computation and communication steps* by the devices—for instance, coordination, management, and self-organization processes.

In particular, we target the idea of programming the overall behavior of such systems by expressing a high-level goal—for instance, 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 a lower level *middleware* or *platform* deal with the complexity of issues like handling dynamicity (e.g., due to mobility or openness), failure, heterogeneity, and so on.

As an analogy, consider database manage-

ment systems (DBMSs): queries express what data has to be retrieved, and the query optimizer attempts to determine the most efficient query plan for satisfying the request, exploiting details such as indexes and physical access methods. The same concept has also been ported to distributed settings: Map-Reduce and Apache Spark, for instance, provide APIs for processing big data on a cluster with implicit data parallelism, fault tolerance, and load balancing. Another example is given by wireless sensor network (WSN) macroprogramming approaches [3], where declarative queries are 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

In general, a self-organizing IoT system should autonomously determine *what* has to be done, together with *when*, *where*, by *whom*, and *how*. In other words, the critical problem is setting up a *decentralized process for adaptive situation recognition and situated action*. The idea is that the system should organize to properly monitor the environment for events or situations requiring intervention; then, the intervention should be organized and carried out to pursue the desired state of affairs. These two phases do not need to be sequential but can be performed continuously in a feedback loop, gradually steering the system towards a correct and stable configuration. Also, the system should *opportunistically* exploit available resources accordingly to the current context and goals—which may change dynamically.

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 intensity of precipitations to prealert the public safety organizations of the areas with high risk of flooding. The phenomenon that we are tracking is spatially and temporally hard to predict with a very fine grain (data from the NOAA[1] reaches, at best, zip-code granularity): at a single-city level, we could perform better by promptly reacting to specialized sensors readings. However, a point of view restricted to a single sensor provides too fragile information, as the condition depends not just on the sensor-local status but the surroundings as well (for instance, a coastal zone of a city with a steep elevation profile could suffer flooding issues even with light rain, if the close by higher-altitude zone is being hit hard). Pre-defining areas (possibly, with knowledge of the city altimetry and structural characteristics) helps, but this strategy still misses out on essential information: how the underlying phenomenon is behaving. Indeed, areas should consider how the city is structured and how the rain is distributed, build ad-hoc sensing areas, and rely on them to perform on-the-fly situation recognition and apply local responses.

This approach to decentralized situation recognition is practical whenever there are one or more phenomena with non-strictly-local effects, when it is spatially shaped irregularly, and/or when it is hard to anticipate its behavior with a sufficiently fine grain.

Requirements and abstractions

Given the high-level vision and goals discussed in the previous sections, and with the help of FLOODWATCH, we delineate some requirements for a programming model for situation recognition and action in IoT systems.

R1. Support for 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. One may also devise a more complex case study and platform for environmental monitoring where there is a distributed process for FLOODWATCH, another process for critical infrastructure monitoring, waste management, surveillance, etc.; these processes may run independently or, possibly, interact.

In general, any complex system is not limited to a single activity but usually involves several activities running concurrently. Furthermore, such activities could be collective, i.e., involve an ongoing collaboration of possibly many agents who may perceive the environment only partially.

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

We call these *concurrent collective tasks (CCTs)*, which express activities that may *overlap* in the system, which means that a device may participate in multiple CCTs simultaneously. Also, notice that CCTs may have a limited and dynamic domain: a subset of devices in the system (sometimes also called *team* or *ensemble*) which may change over time.

R2. Support for 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 sub-optimal: 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.

In many systems, indeed [4], there is a need for abstractions capturing an "adaptive" *spatial divide-and-conquer* principle through which a problem in space (e.g., situation recognition on a smart city) is split into parts (or regions) that dynamically and opportunistically adapt according to context and partial results. 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 progress of activities.

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

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

Related problems and state-of-the-art approaches

The declarative and macro-level 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, hiding details of the low-level constructs of their core languages [5].

The approaches above provide a means for expressing *collective* tasks and processes [8], [2] performed by *ensembles* [9] of devices. From [2], we reuse the idea of computations that spring out, spread across the system, shrink, and vanish to express transient collective operations—simplifying it by hiding to the programmer the complexity of propagation dynamics and structuring their usage by combination with exclusive domains.

The programming model covered in this article is also related to *self-organizing coordination regions (SCR)* [4], a design pattern aiming to solve distributed sensing and load balancing issues in large-scale systems through a dynamic feedback-regulated regional partitioning process. The reader can refer to [4] for a comprehensive survey of works and domains (ranging from WSNs and IoT to edge computing and swarm robotics) exploiting various forms of regional partitioning. More generally, the importance of structured communities in multi-agent systems is witnessed by the sheer number of *organizational paradigms* [10]. However, the programming model provided in this paper sits at a higher

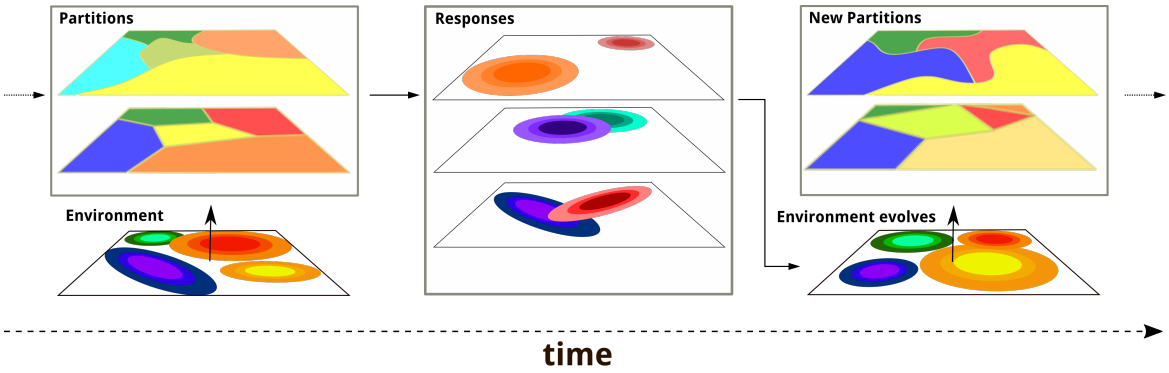

Figure 1: Overview of the proposed approach. The projected squares represent environments, with colors denoting environmental phenomena in terms e.g. of measured data or computed data associated to spatial locations. Elements in gray boxes are software representations, whereas those outside denote real-world spatial 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) responses that can potentially cover only part of the space, overlap, or compete. These responses or the natural environment evolution may lead to a change in the environment, which is tracked by evolving (reshaping, deleting, or creating) the partitions.

level of abstraction and may be seen as exploiting SCR and organizational patterns at the implementation level to solve the problem of decentralized situation recognition and action.

2. Dynamic Decentralization Domains in Practice

From the previous discussion, further refining requirements, we start extrapolating the design elements of an API capable of supporting the kind of 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, through configurable policies, to improve the way the underlying phenomenon is being tracked;
  - the policy to build a sensing domain can be based on a *custom metric*, which can be other than the spatial distance;
- concerning partitioning into DDs (cf. R2) activity within a DD (cf. R3)
  - sensing domains for a single measure will not overlap, to avoid duplicate sampling (and since overlapping can be achieved through multiple CCTs);
  - inside a single sensing domain, a strategy is defined to collect single sensor readings;
  - 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.

From these requirements, we design a Scala API 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], [11], which will be presented and used in the experiments in next section.

```
/* Result produced by
   a distributed sensing operation */
case class SensingResult[D: Numeric](result: D)

/* Configuration of
   a distributed sensing procedure */
class DistributedSensing[
    // areas are centered in higher values
    S: Ordering,
    // distance metric
    D: Numeric,
    // measurement unit of the phenomenon
    V: Numeric
```

```
](
    perceptionCenter: () => S,
    localValue: () => V,
    metric: V => D, // how to consider the distance
    accumulate: UnivariateStatistics[V],
    limit: D // maximum radius of a sensing area
) { def compute(): SensingResult[D] = /*...*/ }

/* An action to perform in
   response to the current state */
type Action[D] = Set[SensingResult[D]] => Unit

/* Abstraction of sensors created
   from decentralized domains */
type DistributedSensors[D] =
  Map[String, DistributedSensing[?, D, ?]]

/* Actual mapping from sensor results and actions
   that will be performed by the nodes */
type SituatedRecognition[D] =
  Map[String, SensingResult[D]] => Set[Action[D]]

// Actual high.level API
def decentralizedSituationRecognitionAndResponse[
  D: Numeric
](
    distributedSensors: DistributedSensors[D],
    situatedRecognition: SituatedRecognition[D]
): Unit = /*...*/
```

Consider the FLOODWATCH case study introduced in Section 1 as a reference scenario. Let us assume that we have several pluviometers deployed in the city capable of communicating with each other (either because they send data to the cloud where some elaboration occurs or because they can communicate directly). We want to monitor the progression of a storm hitting the city, adjusting the granularity at runtime: if in large areas there are similar values, they should get clustered together; if, instead, the precipitation is spotty, we want each spot to track its intensity.

We also know that lower parts of the city are at a higher risk in case of floods. We assume the rain gauges are equipped with a GPS sensor supporting altimetry measurements. Even though altimetry changes are much slower than changes in precipitation intensity, we want to consider both information when responding to a potential emergency. Also, we want to consider the altimetric profile of an entire zone and not of a single point, and we want to react promptly if some 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: we want to put the information to good use, and when the rain in low-altitude areas is so heavy that it might cause floods, we would:

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.

By leveraging the previously shown API, the application logic could be captured by the following code[2], in which we leverage the SCAFI Scala domain-specific language (DSL) [11] to compactly express information-sharing operations (**nbr** shares the provided value with neighboring devices and returns the value in other nodes; `nbrRange` is a shorthand for an **nbr** call measuring the distance from neighbors).

```
def rainGauge(): Double = sense("rain gauge")
def altitude(): Double = sense("gps").altitude
def mean[
  V: Numeric
]: UnivariateStatistics[V] = /*...*/

val waterLevelArea = DistributedSensing(
  rainGauge,
  rainGauge,
  l => nbrRange() * (1 + math.abs(l - nbr(l))),
  mean,
  1e4,
)
val altitudeArea = DistributedSensing(
  altitude,
  altitude,
  h => math.hypot(nbrRange(), h - nbr(h)),
  mean,
  1e4,
)
decentralizedSituationRecognitionAndResponse(
  Map(
    "waterLevel" -> waterLevelArea,
    "elevation" -> altitudeArea,
  ),
  readings => {
    /* prepare the actions
       to perform based on readings */
    Set(
      _ => /*...*/, // propagate alert
      r => if (readings("altitude").result < 20 )
          /*...*/ // find a fire station to prealert
        else /*...*/
    )
  }
)
```

Notice how we use two different metrics and central positions for the two spatial phenomena under observation, and how these data are subsequently used to generate the adaptation strategy at runtime. As described in detail in [11], [2], this specification is also the "script" that each device executes repeatedly in asynchronous *sense–compute–interact* rounds (cf. aggregate execution model), progressively building the intended global behavior, as exemplified in next section.

[2]The full source is available at the repository provided in Section 3.

3. Evaluation

Experimental setup

The exemplar of the previous section has been exercised in a more challenging and realistic scenario. We used the data openly available for the city of Toronto[3], featuring 50 water gauges samples taken in 2021. To challenge our proposed system with a denser network of devices, we added in simulation 300 additional virtual gauges, randomly positioned within the map, whose data is interpolated from the values of the surrounding real devices. We selected the rain event that occurred on 2021-09-07, as it was the heavier in the available data. We used data from OpenStreetMap[4] to position 24 fire stations.

We implemented the proposed API in the ScaFi aggregate programming toolkit [11] and simulated the scenario using the Alchemist simulator [12]. In the experiment, devices compute their programs unsynchronized at a frequency of 1Hz; we measure how many alerts get generated with time and how many devices they reach. Additional gauges position and device timing drift are randomized; we thus ran 64 repetitions of the simulation and considered the mean results. The experiment has been designed to be accessible and reproducible; it has been released as open source[5] and archived for future reference [13]. Figure 2 depicts the scenario as simulated in Alchemist.

Results and discussion

It is evident from Figure 2 that, when conditions change, DDs adapt: they change their shape and extension to track the underlying phenomenon coherently; in response to heavy rain, close-by stations get appropriately alerted.

Figure 3 shows that the system at the macroscopic level tracks the underlying phenomena: more operators get alerted when 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.

[3]https://open.toronto.ca/dataset/rain-gauge-locations-and-precipitation/

[4]using Overpass API https://overpass-turbo.eu/

[5]https://github.com/cric96/experiment-2022-ieee-decentralised-systems

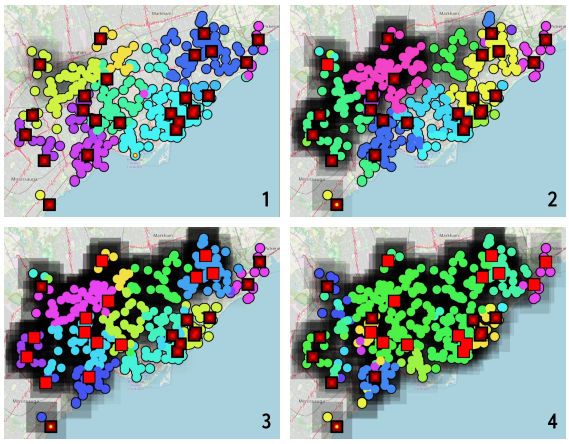

Figure 2: Subsequent simulation snapshots (left-to-right, top-to-bottom) of FloodWatch, simulated in the city of Toronto. Darker shadows indicate heavier rain. Black squares with a red center are firefighters and civil protection centers; their states switch from unalerted (small red dot) to alerted (large red square) when reached by at least one alert. Dots represent gauges; their colors map the DDs they are subject to when measuring the rainfall intensity. At https://doi.org/10.6084/m9.figshare.19626402 there is the video of a complete simulation run.

We now give some remarks to better contextualize the contribution. Regarding applicability and generality, we observe that CCTs and partitioning into DDs enable to address several kinds of applications in domains like ICT ecosystems, WSNs, IoT and smart city, and multi-robot/multi-agent systems—cf. the surveys in [4], [5]. Concerning quantitative cost/performance considerations on this kind of paradigm, details can be found in [4], [2]: the focus of this article is mainly on programming abstractions for decentralized systems, following a language-based software engineering approach [14]. We observe, however, that the dynamic and limited domains of CCTs, and the flexible partitioning into DDs, provides a way to control the level of decentralization and hence trade-offs considering e.g. scalability, reactivity, and monitoring/control granularity. A deep discussion of related work is also beyond the scope of this paper: the interested reader can look at Section 1 for the main references.

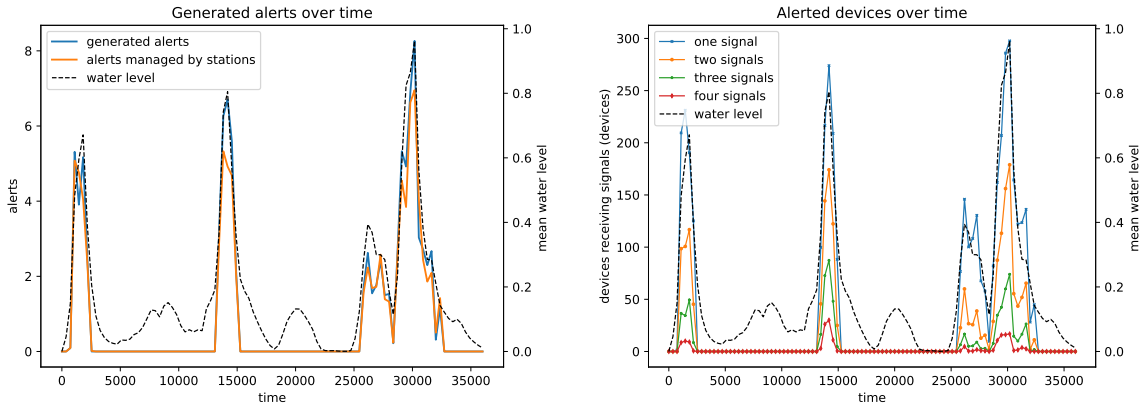

Figure 3: Simulation results, showing, with time, the average rainfall intensity (dotted black line), the number of operator stations receiving alerts (left) and the breakdown by number of alerts received per station (right).

4. Conclusion and Outlook

Mechanisms based on decentralization and self-organization are subject to intense research and are expected to play a crucial role in next-generation applications running on top of 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 providing a structure for the application logic of services that involve (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, FloodWatch. Results show that programs expressed declaratively through the API yield DDs that can adapt to handle distributed monitoring and action properly.

This work has focused on the design and programming of 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 programming abstractions for this class of systems are still quite fragmented. Further, their integration with automatic design approaches, e.g., based on machine learning or program synthesis, may be a fertile path for future research.

Acknowledgment

This work was supported by the MIUR PRIN 2017 Project "Fluidware" (N. 2017KRC7KT) and the MIUR FSE REACT-EU PON R&I 2014-2022 (N. CCI2014IT16M2OP005).

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

Roberto Casadei is a post-doc researcher and adjunct professor at Alma Mater Studiorum–Università di 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, aggregate computing, self-* systems, and IoT/CPS. He has served as workshop chair for the eCAS workshop, as organising and PC member of multiple conferences including COORDINATION and ACSOS, as reviewer for renowned international journals. He also leads the development of the open-source ScaFi aggregate programming toolkit.

Danilo Pianini is a post-doctoral researcher at DISI, the Department of Computer Science and Engineering of the University of Bologna, in Italy. He holds a PhD in Computer Science and Engineering, and his main research interests include simulation, (self-organising) coordination, aggregate computing, pervasive systems, software engineering, agile techniques, and DevOps. He published over 50 articles in international journals and conferences on those subjects. He is the lead designer of dozens of open-source software tools, including the Alchemist simulation platform and the Protelis aggregate programming language.

Mirko Viroli is Full Professor in Computer Engineering at the University of Bologna, Italy. He is an expert in foundations of computer science and programming, object-oriented programming, advanced software development, software engineering and self-adaptive/self-organising pervasive computing systems. He is author of more than 300 papers, of which more than 70 on international journals. His GoogleScholar h-index is 47 with ¿7000 citations. He is member of the Editorial Board of IEEE Software magazine, and was program chair of the ACM Symposium on Applied Computing (SAC 2008 and 2009), and IEEE Self-Adaptive and Self-Organizing systems (SASO 2014) conferences.

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. 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, 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
12. 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
13. G. Aguzzi and D. Pianini, "cric96/experiment-2022-ieee-decentralised-system: 1.0.1," 2022. [Online]. Available: https://zenodo.org/record/6477039
14. 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