

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Towards Pulverised Architectures for Collective Adaptive Systems through Multitier Programming

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:

Aguzzi G., Casadei R., Pianini D., Salvaneschi G., Viroli M. (2021). Towards Pulverised Architectures for Collective Adaptive Systems through Multitier Programming. Los Alamos, CA : Institute of Electrical and Electronics Engineers Inc. [10.1109/ACSOS-C52956.2021.00033].

Availability:

This version is available at: <https://hdl.handle.net/11585/875832> since: 2022-03-01

Published:

DOI: <http://doi.org/10.1109/ACSOS-C52956.2021.00033>

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (<https://cris.unibo.it/>).
When citing, please refer to the published version.

(Article begins on next page)

This is the final peer-reviewed accepted manuscript of:

G. Aguzzi, R. Casadei, D. Pianini, G. Salvaneschi and M. Viroli, "Towards Pulverised Architectures for Collective Adaptive Systems through Multi-Tier Programming," 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), DC, USA, 2021, pp. 99-104

The final published version is available online at <https://dx.doi.org/10.1109/ACSOS-C52956.2021.00033>

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (<https://cris.unibo.it/>)

When citing, please refer to the published version.

Towards Pulverised Architectures for Collective Adaptive Systems through Multi-tier Programming

Gianluca Aguzzi*, Roberto Casadei*, Danilo Pianini*, Guido Salvaneschi[†] and Mirko Viroli*

* ALMA MATER STUDIORUM—Università di Bologna, Cesena, Italy
Email: {gianluca.aguzzi, roby.casadei, danilo.pianini, mirko.viroli}@unibo.it

[†]University of St.Gallen: St.Gallen, Switzerland
Email: guido.salvaneschi@unisg.ch

Abstract—Engineering large-scale *Cyber-Physical Systems* – like robot swarms, augmented crowds, and smart cities – is challenging, for many issues have to be addressed, including specifying their collective adaptive behaviour and managing the connection of the digital and physical parts. In particular, some approaches propose *self-organising* mechanisms to actually *program* global behaviour while fostering decentralised, asynchronous execution. However, most of these approaches couple behavioural specifications to specific network architectures (e.g., peer-to-peer), and therefore do not promote flexible exploitation of the underlying infrastructure. Conversely, *pulverisation* is a recent approach that enables self-organising behaviour to be defined independently of the available infrastructure while retaining functional correctness. However, there are currently no tools to formally specify and verify concrete architectures for pulverised applications. Therefore, we propose to combine pulverisation with multi-tier programming, a paradigm that supports the specification of the architecture of distributed systems in a single code base, and enables static checks for the correctness of actual deployments. The approach can be implemented by combining the ScaFi aggregate computing tool-chain with the ScalaLoc multi-tier programming language, paving the path to support the development of self-organising cyber-physical systems, addressing both functional (behaviour) and non-functional concerns (deployment) in a single code base and modular fashion.

Index Terms—Pulverisation, Aggregate Computing, Multi-tier programming

I. INTRODUCTION

Current socio-technical developments foster a vision of worlds populated by large-scale Cyber-Physical Systems (CPSs)—such as robot swarms, crowds of augmented people, and ecosystems of smart devices. Such systems are typically required to act as coordinated collectives and to be able to adapt to dynamic environmental conditions and inputs—i.e., they are meant to be Collective Adaptive Systems (CASs). However, engineering CASs is a challenging task. Generally, reliance on a centralised device is not allowed, and the system is expected to reach global goals based on the coordinated (inter-)actions of the individual entities that compose the system. Practical solutions, such as the Buzz language [?], use a so-called “meet-in-the-middle” method by which developers can use individual and global viewpoints (through `swarm` operator) as needed. Recent approaches such

as aggregate computing [1], [2], DEECo [3], SmartSociety [4], and attribute-based coordination [5], foster the adoption of *global-to-local* techniques, by which the behaviour of the ensemble of devices is designed top-down, and interactions among devices (i.e., protocols) are generated automatically and implicitly. However, the generated interaction scheme depends on assumptions on how devices communicate with each other; in other words, the approach often dictates how the network should be structured. In turn, this creates a tension between the network structure the language reasons upon and the actual way devices communicate. Since in CASs the challenging case in which no controllers exist must typically be supported, a purely *peer-to-peer* (P2P) network is usually considered the paradigmatic setup. However, real-world networks are usually structured hierarchically, and the ability to target multiple infrastructural setups can help to achieve non-functional benefits.

Pulverisation [6] is an approach proposed for aggregate computing (but in principle applicable to other frameworks) to neatly separate behavioural and deployment concerns. In short, it decomposes the concept of a *logical device*, which is the target for which the behaviour is programmed, into micro-components that can be deployed independently and whose internal communication protocol is defined at deployment time. This technique de-facto relieves the behaviour designer from the duty to consider multiple possible deployments in different networks, allowing them to write the behaviour for the most generic case, and have the program *functionally* behave as designed regardless of the actual final deployment.

However, pulverisation does not directly provide ways for *specifying* and *deploying* components in a safe and meaningful fashion: it proposes a methodology to cleanly separate behavioural and deployment concerns that need to be addressed at some point. The definition of the deployment strategy and its execution is thus a very relevant and challenging engineering issue on its own: ideally, such a specification should be declarative and possibly guided and checked by static analysis to lower the risk of failures at runtime.

This work discusses the initial research effort to fill the gap between a pulverised system and its actual deployment and execution on multiple different network structures, by leveraging a recent approach known as *multi-tier programming* [7]. In

multi-tier programming, a distributed system is *declaratively* described, in terms of components and admissible interactions *in a single code base*. In particular, in type-level multi-tier programming, the specification leverages the type system of the language to ensure the correctness and coherence of the architecture; moreover, it also relieves the developer from low-level concerns by offloading to the compiler the responsibility of breaking the computation into deployment units enforcing the contract defined in the code.

The remainder of this paper is as follows: Section II introduces the current state of the art, discussing pulverisation in the context of aggregate computing and multi-tier programming; Section III discusses the possible integration of pulverised aggregate computing systems and multi-tier programming and shows initial experiments leveraging the ScaFi and ScalaLoci Scala domain-specific languages (DSLs); Section IV exposes and discusses the short and long term implications of this integration; finally, Section V concludes and discusses the next research steps.

II. BACKGROUND

A. Pulverised aggregate computing

At the core of pulverisation [6] is the idea that the functional behaviour of a distributed application is fundamentally orthogonal to the actual deployment of the services that compose it. Thus, through a classic *divide-and-conquer* approach, in a pulverised system, any *logical device* (of the many composing the CAS) is broken down into five *components* acting as *units of deployment*: 1) *Sensors (S)*, encapsulating the ability to retrieve information from the environment; 2) *Actuators (A)*, responsible for acting upon the environment; 3) *State (K)*, providing persistence of knowledge; 4) *Behaviour (B)*, modelling the actual execution of the application business logic; and 5) *Communication (C)*, which provides means to interact with other logical devices. These pulverised components can be deployed to different physical nodes of the network: as far as they can communicate with each other and the target execution protocol is respected, the functionality should not be affected. Then, a concrete development approach will expose abstractions with a well-defined mapping to such a partitioning schema. So, an application designer can focus on functional requirements while delaying all the deployment and communication concerns (which may well affect non-functional properties of the system) to a later moment.

This strategy is especially well-suited to adapt approaches designed to work with a flat (non-layered) network structure (e.g., peer-to-peer, mesh, and ad-hoc networks) to arbitrary network architectures—to exploit a broader range of deployments, e.g. for efficiency or reliability. Consider, for instance, the simple case of Figure 2a: there is a 1:1 mapping between logical and physical devices, and direct communication among devices (actually, among their C pulverised components) must be possible. This is typically not the case in many Internet applications, however: let us consider the case in which the same application should be deployed in an IoT scenario where end devices are *thin*, equipped with sensors and actuators,

but battery-powered and equipped with a microcontroller with minimal computational capabilities (for instance, LoRaWAN or Sigfox motes [8]). These devices cannot host the actual computation of the program (component B), which must necessarily be offloaded to the edge or the cloud. This change in the deployment would typically imply a re-writing of the functional logic of the program, as end devices cannot be considered computation-capable nodes any more. Instead, with pulverisation, they retain their existence as logical devices with some of their pulverised components hosted on different physical nodes as depicted in Figure 2c. Crucially, this makes the original application work on a different network architecture without any functional logic changes.

Aggregate computing [1], [2] is a global-level functional programming model for CAS where a single program expresses the collective adaptive behaviour of a distributed system as a whole. Also, aggregate computing is a naturally pulverisable approach: its semantics can be expressed as a purely functional manipulation of state, messages, and sensor readings [2], providing a straightforward mapping into pulverised components. Indeed, initial experiments [6] showed that aggregate programs deployed on pulverised infrastructures retain their original functional behaviour onto different deployments. Nevertheless, pulverisation is not a silver bullet: the approach is fundamentally an engineering pattern to encapsulate a non-functional concern (network structure and deployment), allowing for the business logic to work across deployments, but it does not specify *how* a pulverised architecture should be described and verified so that it can be operated correctly at runtime.

B. Multi-tier programming and ScalaLoci

The concrete architecture of a distributed system is usually *multi-tier*, i.e., it comprises multiple layers, each one encapsulating some specific functional concern (e.g. data management, application and presentation logic, etc.) each physically separated from the others. Historically, distinct tiers and crosscutting functionalities that belong to multiple tiers are developed into several compilation units (often using different programming languages), raising development and maintenance costs.

A recent trend trying to tackle these issues is *multi-tier programming* [7], by which a distributed architecture is defined in a single compilation unit with a single language. Once the program is declaratively specified, the compiler (or the runtime, depending on the language of choice) is responsible for splitting the computation among different peers. Depending on the specific multi-tier programming language, different kinds of constraints may be imposed. For instance, in Links [9], applications must follow a client-server architecture, while other languages allow for more freedom of choice.

One interesting language that lets the designer specify arbitrary deployments is ScalaLoci [10]–[12], a type-safe multi-tier language hosted in Scala language. The structure of a ScalaLoci application is defined through *peers* and *ties*. Peers abstract over locations and represent the components of an

application, whereas ties define the connections between peers. Only tied peers can communicate with each other.

The following code depicts a simple controller-worker architecture. Annotation `@multitier` denotes the `BookingApp` as a ScalaLoc object.

```
/* Defines an application with the peers 'Controller' and
 * and 'Worker' and a 1:n connection between them */
@multitier object BookingApp {
  @peer type Controller <: {
    type Tie <: Multiple[Worker]
  }
  @peer type Worker <: {
    type Tie <: Single[Controller]
  }
  on[Controller]{ print("I am a Controller") }
  on[Worker]{ print("I am a Worker") }
}
```

A declared `@peer` type can have multiple instances that execute the peer's logic, e.g., multiple worker instances. In this example, the logic is replaced with simple prints. An instance of the controller peer may connect to multiple workers, whereas a worker instance is tied to one controller. The sample compiles two executables representing the controller and the worker, whose instances can be deployed and executed on different physical nodes.

```
/* accessible for workers. */
val requests: Event[Request] on Controller = placed {...}
// Name of the worker @Worker accessible for Controller.
val name: String on Worker = placed {...}
/* not accessible for workers. */
val tokens : Local[Map[Long]] on Controller =
  placed {...}
/* Access allowed: Worker observes events
  emitted on Controller. */
on[Worker]{ requests.asLocal.observe {...} }
// Error: no access to tokens outside of the Controller.
on[Worker]{ tokens.asLocal.observe {...} }
```

Asynchronous multi-tier reactivities like signals and events are used to compose non-blocking data flows that span across multiple peers. Data from remote peers are accessed using ScalaLoc's `.asLocal` expression variants, and the visibility of placement types for remote peers can be regulated. A `@multitier` module can capture the controller-worker schema:

```
@multitier trait ControllerWorker[T] {
  @peer type Controller <: {
    type Tie <: Multiple[Worker]
  }
  @peer type Worker <: {
    type Tie <: Single[Controller]
  }
  def run(task: Task[T]): Future[T] on Controller =
    // run task on some selected worker
    on(selectWorker()) // ('selectWorker' is left out)
    .run.capture(task) { task.process() }.asLocal
}
```

The `run` method has return type as the placement type `Future[T]` on `Controller`¹, effectively placing `run` on

¹Scala enables infix use of binary type constructors; i.e., `A on B` refers to the same type as `on[A, B]`.

the `Controller` peer. The `Task` type is parametrised over the type `T` of the value, which a task produces after execution. Running a remote task remotely results in a `Future` to account for processing time and network delays and potential failures. The remote block is executed on the worker, which starts processing the task. The remote result is transferred back to the controller as `Future[T]` using `asLocal`. A single worker instance in a pool of workers is selected for processing the task via the `selectWorker` method.

The module can be used to implement an application where a server offloads work to the connected clients. In the following code, we specialise the clients to be workers and the server to be a controller:

```
@multitier trait VolunteerProcessing {
  val m: ControllerWorker[Int] // ref to another module
  // augmenting the peers in this module
  @peer type Client <: m.Worker
  // with the controller/worker functionality
  @peer type Server <: m.Controller
  on[Server] { m.run(new Task()) }
}
```

III. MULTI-TIER PULVERISED AGGREGATE COMPUTING

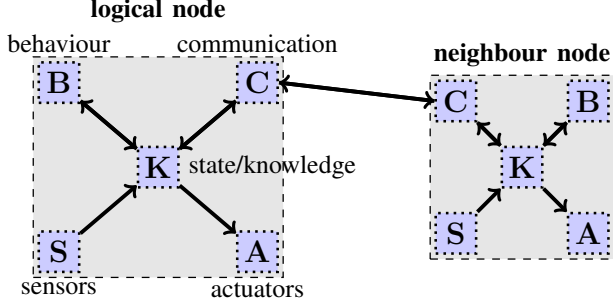
The contribution of this work is an architecture for multi-tiered deployment strategies in pulverised systems, along with a prototypical implementation using aggregate programming and ScalaLoc. Using multi-tier abstractions, we:

- 1) map the overall logical system into a multi-tiered module, building the concept of pulverised device into ScalaLoc (see Figure 1b);
- 2) define the functions associated with each pulverised component;
- 3) characterise the possible kinds of network nodes (e.g., cloud, edge, thin end device);
- 4) decide the network structure in terms of possible connections among network node kinds;
- 5) detail the deployment by assigning each pulverised component to a network node kind.

Ultimately, this architectural design allows us to specify functional behaviour independently of deployment (via pulverised aggregate programming), then declaratively define multiple deployment schemes and their related communication constraints (thanks to multi-tier programming), and finally, statically enforce the respect of the expressed constraints (as a consequence of the robust type programming system introduced by ScalaLoc).

A. Pulverised architecture in ScalaLoc

As a first step, we need to formalise what a pulverised architecture is in ScalaLoc, by defining all the pulverised components and binding them together into the concept of logic node. Figure 1 shows a possible ScalaLoc implementation (Figure 1b) of a pulverised device (Figure 1a): `LNode` represents the logical device, `LogicalSystem` encloses the concept of pulverised system into a multi-tier module. The



(a) A pulverised logical device, split into sub-components, and one of its neighbours.

```
@multitier trait LogicalSystem {
  // A logical node, connected to other logical nodes
  @peer type LNode <: { type Tie <: Multiple[LNode] }
}

// Partitioning of a logical node into sub-components
@multitier trait PulverisedSystem extends LogicalSystem {
  @peer type SensorComponent <: LNode // S
  @peer type ActuatorComponent <: LNode // A
  @peer type StateComponent <: LNode // K
  @peer type BehaviourComponent <: LNode // B
  @peer type CommunicationComponent <: LNode // C
}
```

(b) ScalaLoc code describing a pulverised logical system. (See Section III-C for more details.)

Fig. 1: Pulverisation model and corresponding ScalaLoc specification.

logical device and all its pulverised components are mapped on abstract peers.

Once all components are modelled, their contract must be specified to characterise them and define their behaviour. This is done by *placing* the available computations on the components that will effectively host them. For instance, if our system has the notion of `Sensor[V]`, representing a generic sensor that upon access returns values of type `V`, we can enforce the requirement that the `SensorComponent` must be able to read values from sensors via something like:

```
def sense[V](id: SensorID): V on SensorComponent = ...
```

This strategy decouples the *structural* definition of components participating in the system from their *behavioural* specification.

B. Definition of deployment kinds

Once the definition of components is complete, we can begin describing the actual deployments. These can be expressed rather concisely with the proposed design, as depicted in Figure 2, where we show three possible definitions of very different architectures. In Figure 2a, we define a system where logical and physical devices coincide. This structure is typical of opportunistic network structures (P2P overlays, tactical networks, etc.). Figure 2b, shows a hybrid edge-cloud system supporting the computation of *thick* end devices (e.g., smartphones). The infrastructure hosts the communication components, de facto enabling network communication among end devices (this is a typical situation in usual WiFi networks, where end devices are “hidden” a router performing network address translation). A practical example of this architecture could be a multi-broker MQTT system, with brokers deployed either on the edge (for better performance with closely located devices) or on the cloud. Finally, in Figure 2c, we replicate a similar system, but with *thin* end devices. Namely, end devices

do not possess enough computational capacity to host their associated computation and thus need to operate as remote sensors and offload all calculations to an external device. This situation is typical of WAN sensing networks (e.g., LoRaWAN), where end devices are equipped with minimal memory and very low power microcontrollers and are expected to run on battery for years. To summarise, different network architectures can be specified by following two steps: 1) definition of the physical devices involved in the architecture and how they are *tied* together; and 2) allocation of the pulverised components on the kinds of devices that can host them.

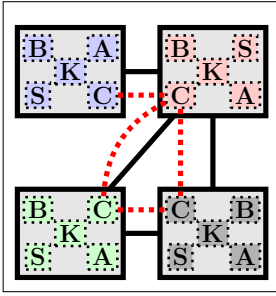
The resulting system can then be instantiated by selecting a communication protocol and a serialisation framework. For example, in the following snippet, we show how this could be done for the system in Figure 2b, assuming communication via TCP and serialisation via the uPickle library.

```
import loci.serializer.upickle._ // Serialization logic
import loci.communicator.tcp._ // Communication protocol
object Broker extends App { // Peer instantiation
  val tie = listen[BrokerBased.Peer](TCP(port))
  multitier.start(new Instance[BrokerBased.Broker](tie))
}

object Peer extends App {
  val tie = connect[BrokerBased.Broker](TCP(host, port))
  multitier.start(new Instance[BrokerBased.Node](tie))
}
```

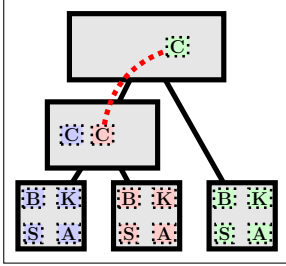
C. Integration with Aggregate Programming

The design described so far is entirely independent of the specific aggregate programming language of choice: due to pulverisation, the way the logic is expressed only concerns the behavioural component (B). Currently, there are three choices for practical aggregate programming: Protelis [13], a stand-alone, JVM-hosted domain-specific language (DSL); FCPP [14], a high-performance, low-memory footprint C++ implementation; and ScaFi [?], a Scala internal DSL that can run on the JVM or in the browser [16]. Among the three, we



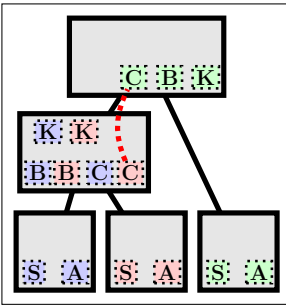
(a) Peer-to-peer: 1-to-1 mapping between logical and physical devices.

```
@multitier object P2P extends PulverisedSystem {
  // Definition of device kinds and possible network connections
  @peer type Node <: { type Tie <: Multiple[Node] }
  @peer type SensorComponent <: Node
  // Pulverised component allocation on devices
  @peer type ActuatorComponent <: Node
  @peer type StateComponent <: Node
  @peer type BehaviourComponent <: Node
  @peer type CommunicationComponent <: Node
}
```



(b) Multi-broker: the communication is offloaded in part to the edge, and in part to the cloud.

```
@multitier object BrokerBased extends PulverisedSystem {
  // Definition of device kinds and possible network connections
  @peer type Node <: { type Tie <: Single[Broker] }
  @peer type Broker <: { type Tie <: Multiple[Node] with Multiple[Broker] }
  // Pulverised component allocation on devices
  @peer type SensorComponent <: Node
  @peer type ActuatorComponent <: Node
  @peer type StateComponent <: Node
  @peer type BehaviourComponent <: Node
  @peer type CommunicationComponent <: Broker
}
```



```
@multitier object IoTSystem extends PulverisedSystem {
  // Definition of device kinds and possible network connections
  @peer type Thin <: { type Tie <: Multiple[Thick] }
  @peer type Thick <: { type Tie <: Multiple[Thick] with Multiple[Thin] }
  // Pulverised component allocation on devices
  @peer type SensorComponent <: Thin
  @peer type ActuatorComponent <: Thin
  @peer type StateComponent <: Thick
  @peer type BehaviourComponent <: Thick
  @peer type CommunicationComponent <: Thick
}
```

(c) IoT with thin clients: end devices only host sensors and actuators, other components are offloaded either to the edge or the cloud.

Fig. 2: Examples of pulverised architectures. Thick boxes represent physical devices, dashed boxes represent pulverised components, and different logical devices are identified by colour (red, green, and blue). A pulverised component is hosted on the physical device in which it is contained. Communication among different logical devices that imply communication among physical devices is depicted with a dashed red line.

picked ScaFi for our prototype, mainly because it shares the language of choice with ScalaLoc, and thus it could be the foundation stone of a unified framework living in the Scala ecosystem. A full account of ScaFi can be found in [15].

In order to perform a collective computation, ScaFi requires to define an **AggregateProgram** (i.e. an object containing the aggregate application logic) and a **Context** (i.e., the set of information required to evaluate an **AggregateProgram**, such as the previous state, sensors' data, and messages received from neighbours). ScaFi's **Context**s in a pulverised architecture are embedded in the **State** component. Consequently, the glue code required to execute ScaFi aggregate code over a pulverised network is minimal:

```
def compute(
  deviceIdentifier: Id,
  state: State
): State on BehaviourComponent = {
  val context = new ContextImpl(
    deviceIdentifier,
    export = state.exports,
    localSensor = state.sensors,
    neighbourSensor = state.neighbourSensor
  )
  val program : AggregateProgram = ... // business logic
  // actual execution; returns the new State
  program.round(context)
}
```


IV. IMPLICATIONS

The construction of a pulverised platform for aggregate computing through multi-tier programming has several interesting applications that we analyse in this section.

Programmability and compile-time safety of deployment architectures for pulverised systems: A type-annotated definition of a pulverised system deployment can be used to statically enforce consistency between the intended architecture and its deployment implementation. Access to data that cannot be reached by a certain component will not be possible as it will be checked by the compiler. Indeed, engineering should only concern about functional aspects of the application — namely, write an aggregate computing program. We note that this research direction could be seen as a part of a larger effort to encapsulate the *functional* part of the application (i.e., writing the aggregate computing program), isolating *non-functional* concerns and tackling them separately, possibly through dedicated techniques and languages.

Opportunistic deployment and reconfiguration of pulverised systems: As a matter of principle, the pulverisation approach supports self-adaptive application deployment. Indeed, nothing prevents moving components of a logical device across physical nodes (e.g. B component can be placed into the cloud if the device has a low battery level). This direction, however, is not currently supported by ScalaLoc, which assumes static data placement. The influence of our design could lead to an extension of the languages in which the placed type could be moved data between peers, while still leaving the system specification type-safe.

Placement types in aggregate programming: Currently, ScalaLoc and ScaFi had been *intentionally* combined in such a way that the aggregate program is completely agnostic about the usage of a multi-tier language for deployment. A different yet intriguing research direction would be to understand if and how placement types (along with the other concepts introduced by the peculiar interpretation of multi-tier programming in ScalaLoc) could be leveraged in an aggregate setting. At the moment, the implications of manipulating placement types at the aggregate computing level are unclear, but we glimpse some potential for the definition of evolving networked systems that are worth exploring.

V. CONCLUSION

In this article, we propose an approach to bridge pulverised architectures and verified deployment of aggregate systems, leveraging multi-tier programming to foster declarativity, expressiveness, and safety. In particular, we show how a pulverised architecture could be specified in ScalaLoc, and then mapped onto a concrete deployment. Finally, we show how aggregate computations can be easily soldered into the infrastructure, by drafting the implementation of the behavioural part of the system using the ScaFi aggregate computing toolkit.

Even though the present work is preliminary, we show that the approach is general enough to support different deployment architectures: we exemplify peer-to-peer and IoT edge-cloud architecture with thick or thin end devices, but of course,

the design space in this context is huge. We believe that the integration between multi-tier programming and pulverised aggregate programming is a promising approach for the design and implementation of collective adaptive systems that can execute independently of the underlying infrastructure, preserving the business logic.

Acknowledgements

This work has been supported by the MIUR PRIN 2017 Project N. 2017KRC7KT “Fluidware”.

REFERENCES

- [1] J. Beal, D. Pianini, and M. Viroli, “Aggregate programming for the internet of things.” [Online]. Available: <https://doi.org/10.1109/MC.2015.261>
- [2] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, “From distributed coordination to field calculus and aggregate computing.” [Online]. Available: <https://doi.org/10.1016/j.jlamp.2019.100486>
- [3] T. Bures, I. Gerostathopoulos, P. Hnetyuka, J. Keznikl, M. Kit, and F. Plasil, “DEECO: an ensemble-based component system.” [Online]. Available: <https://doi.org/10.1145/2465449.2465462>
- [4] O. Seckic, D. Miorandi, T. Schiavinotto, D. I. Diochnos, A. Hume, R. Chenu-Abente, H. L. Truong, M. Rovatsos, I. Carreras, S. Dustdar, and F. Giunchiglia, “Smartsociety - A platform for collaborative people-machine computation.” [Online]. Available: <https://doi.org/10.1109/SOCA.2015.10>
- [5] Y. A. Alrahman, R. D. Nicola, and M. Loret, “Programming interactions in collective adaptive systems by relying on attribute-based communication.” [Online]. Available: <https://doi.org/10.1016/j.scico.2020.102428>
- [6] R. Casadei, D. Pianini, A. Placuzzi, M. Viroli, and D. Weyns, “Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment.” [Online]. Available: <https://doi.org/10.3390/fi12110203>
- [7] P. Weisenburger, J. Wirth, and G. Salvaneschi, “A survey of multitier programming.” [Online]. Available: <https://doi.org/10.1145/3397495>
- [8] K. Mekki, E. Bajic, F. Chaxel, and F. Meyer, “Overview of cellular LPWAN technologies for iot deployment: Sigfox, lorawan, and nb-iot.” [Online]. Available: <https://doi.org/10.1109/PERCOMW.2018.8480255>
- [9] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, “Links: Web programming without tiers.” [Online]. Available: https://doi.org/10.1007/978-3-540-74792-5_12
- [10] P. Weisenburger, M. Köhler, and G. Salvaneschi, “Distributed system development with scalaloc.” [Online]. Available: <https://doi.org/10.1145/3276499>
- [11] P. Weisenburger and G. Salvaneschi, “Multitier modules.” [Online]. Available: <https://doi.org/10.4230/LIPIcs.ECOOP.2019.3>
- [12] —, “Implementing a language for distributed systems: Choices and experiences with type level and macro programming in scala.” [Online]. Available: <https://doi.org/10.22152/programming-journal.org/2020/4/17>
- [13] D. Pianini, M. Viroli, and J. Beal, “Protelis: practical aggregate programming.” [Online]. Available: <http://doi.acm.org/10.1145/2695664.2695913>
- [14] G. Audrito, “FCPP: an efficient and extensible field calculus framework.” [Online]. Available: <https://doi.org/10.1109/ACSOS49614.2020.00037>
- [15] R. Casadei, M. Viroli, G. Audrito, D. Pianini, and F. Damiani, “Engineering collective intelligence at the edge with aggregate processes.” [Online]. Available: <https://doi.org/10.1016/j.engappai.2020.104081>
- [16] G. Aguzzi, R. Casadei, N. Maltoni, D. Pianini, and M. Viroli, “Scafi-web: A web-based application for field-based coordination programming.” [Online]. Available: https://doi.org/10.1007/978-3-030-78142-2_18