On Execution Platforms for Large-Scale Aggregate Computing

Mirko Viroli
Università di Bologna
via Sacchi 3, 47521 Cesena,
Italy
mirko.viroli@unibo.it

Roberto Casadei
Università di Bologna
via Sacchi 3, 47521 Cesena,
Italy
roberto.casadei12@studio.unibo.it

Danilo Pianini
Università di Bologna
via Sacchi 3, 47521 Cesena,
Italy
danilo.pianini@unibo.it

Ubicomp/ISWC'16 Adjunct, September 12–16, 2016, Heidelberg, Germany.

ACM 978-1-4503-4462-3/16/09...$15.00
DOI: http://dx.doi.org/10.1145/2968219.2979129

Abstract
Aggregate computing is proposed as a computational model and associated toolchain to engineer adaptive large-scale situated systems, including IoT and wearable computing systems. Though originated in the context of WSN-like (peer-to-peer and fully distributed) systems, we argue it is a model that can transparently fit a variety of execution platforms (decentralised, server-mediated, cloud/fog-oriented), due to its ability of declaratively designing systems by global-level abstractions: it opens the possibility of intrinsically supporting forms of load balancing, elasticity and toleration of medium- and long-term changes of computational infrastructures. To ground the discussion, we present ongoing work in the context of `scafi`, a language and platform support for computational fields based on the Scala programming language and Akka actor framework.

Author Keywords
Aggregate computing, Large-scale systems, Internet of Things, Execution platforms, Cloud computing

Aggregate Computing
Abstraction is one of the main tools that engineers can use to tackle the multi-faceted and -layered complexity of nowaday's scenarios of distributed and situated computation such as Internet of Things (IoT) and cyber-physical

Case study and motivation

Let's consider a multitude of people with their smartphones (or other wearable devices) at a public mass event (e.g., a concert or a fair). All the devices that run an aggregate application constitute an aggregate system, computing the global field of the devices' state, e.g., useful to support crowd steering scenarios [1]. Now, it turns out that certain devices may not be able to (physically) communicate in a peer-to-peer fashion; this motivates server-based or cloud-based execution support. Also, certain areas may become overcrowded, possibly overpowering the local base station, resulting in the loss of mobile web access; this might motivate turning back to p2p execution. Hence, pragmatically, large-scale pervasive computing systems require flexible executing platforms.

systems [4]. A promising approach to constructing a desirable abstraction layer on top of large scale ubiquitous systems is *aggregate computing* [1]. This is based on the idea of shifting from the standard single-device focus on system programming to an aggregate viewpoint in which one sees the overall set of computational devices spread in the pervasive computing environment as a single "machine", a sort of diffused computational *fabric*. The target of computation, then, is the conceptually atomic manipulation of distributed data structures—sensors data ultimately considered as input, actuation as output.

How can these global computations, spread over the whole network of devices, be characterised? A first required ingredient is a model of the "domain" of computation, comprising a topological model of the pervasive environment, the set of situated computational devices, and a notion of context regulating which part of the computational environment can actually affect computation at a device. Second, a model of computation over such domain needs to be established. As our goal is to create programs by specifying how to transform, condense, and propagate these distributed data structures, we observe that any such program necessarily involves a computational process spanning a non-pointwise region of space and time. However, such programs also need to be properly turned into (computation/communication) acts carried on by the single devices.

In the most general settings, such local tasks, which we call *computation rounds*, comprise the following mechanisms: *(i)* gathering of messages from other related/nearby devices, *(ii)* perception of contextual information through sensors, *(iii)* storing local state of computation, *(iv)* computing a new local state, *(v)* dispatching messages to neighbours, and *(vi)* executing some form of actuation. Put in a different way, aggregate computing is about defining global behaviours which can be "pulverised" into a computational "dust", formed by atomic actions (i.e., rounds) to be repetitively executed through space and time by the many devices available. As discussed in the following sections, this will be key to flexibly and transparently map aggregate computations into a variety of execution platforms.

This abstract framework described finds realisation through the notion of *computational field* [6] and the associated *field calculus* [3]. We can consider a computational domain model given by a (possibly dynamic) neighbouring relation representing physical (or logical) proximity. Drawing inspiration from physics, then, a *computational field* is a distributed data structure mapping each device (located in a point of space-time) to a computational object. The field calculus, then, formalises the basic primitives for achieving a global and expressive manipulation of these collective data structures. Thus, global computations are functions from field to field: it is possible to lift traditional functions (e.g., mathematical/computational functions) to work with fields and, most notably, we can identify reusable building blocks to capture recurrent field manipulations—effectively devising a library of field-based computation functions [1].

Field calculus programs are executed by a network of computing devices. Every device is given the same program expressing the global computation, which is later compiled into a computation round function according to a well-defined operational semantics—devices running different programs is actually a specific case, handled as an higher-order concept [3]. The result of a round is called an *export* and is broadcasted to the whole neighbourhood. The local execution context consists of the last

```
trait Constructs {
  def rep[A](init: A)
            (fun: (A) => A): A
  def nbr[A](expr: => A): A
  def foldhood[A](init: => A)
                 (acc: (A,A)=>A)
                 (expr: => A): A
  def branch[A](cond: => Boolean)
               (th: => A)
               (el: => A): A
  def sense[A](name: LSNS): A
  def nbrvar[A](name: NSNS): A
}
```

Figure 1: All the basic primitives of the field calculus naturally turn into methods, and are declared in the `Constructs` trait, which is "mixed in" in any aggregate program.

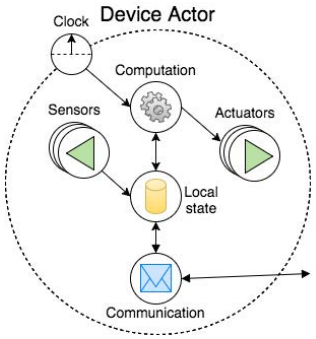

Figure 2: Conceptual model of a device actor in `scafi`: each concern (computation, sensing, actuation, state store and communication) is handled by a different child actor. Different platforms may factor some child actors outside the device.

computed export, the exports received from the neighbours, the local sensors and the "environmental sensors," which are provided by the aggregate programming infrastructure. It is exactly the local context that defines the delta that gives, to each device, a peculiar behaviour that ultimately results into a global, system-wide effect.

With aggregate computing the importance of individual computing devices fades. In [1, 7] several advantages of the approach are discussed, there including the ability of expressing algorithms of collective adaptation as reusable blocks, of structuring computations in a way that is effectively independent of devices number and location, and the intrinsic resiliency to changes in the environment.

In this paper, we address a further feature of aggregate computing: flexible deployment to a variety of execution platforms. We try to provide an answer to the following question: what would be a proper way of abstracting away execution platform issues into pervasive computing services running on top of very large scale systems? We show that aggregate computing smoothly adapts to peer-to-peer fully distributed platforms, server-based ones providing communication and/or computational services, paving the way towards full, dynamic edge- and cloud-based support.

Aggregate system development in Scala

`scafi` (Scala fields) is an aggregate programming framework, developed in the Scala programming language, that includes: *(i)* an internal Domain Specific Language (DSL) providing a syntax and the corresponding semantics for the field calculus constructs as Scala APIs, and *(ii)* a distributed middleware that supports the configuration of aggregate systems, according to different distributed computing scenarios, as well as their runtime execution.

We choose Scala as the host language for building an aggregate programming platform, since it is a modern language for the JVM which integrates the object-oriented and functional paradigms in a seamless way, has a powerful and expressive type system, and supports the development of fluent APIs perceived by users as "embedded languages". Also, Scala's concurrency support and the availability of toolkits such as the Akka actor-based framework make it a valuable tool for building infrastructures for distributed processing systems—see e.g. initiatives like Apache Spark and Storm.

The framework has an implementation of the semantics of the field calculus constructs (see Figure 1), essentially providing a virtual machine for the execution of computational rounds, as more thoroughly described in [3, 1].

The key point is that it is possible to define common operations on fields and high-level behaviours, such as self-organisation patterns and coordination algorithms, by the simple implementation of Scala methods—in fact, the functional character of the approach promotes systematic factorisation of behaviour into building blocks and layers of increasing abstraction [1], all coming with provable resilient properties [7]. On top of such resilient building blocks, sound and expressive APIs can be defined, e.g.: function `randomRegions(d)` (creating a partition of the network in regions of diameter `d`), `average(partition,val)` (averaging `val` in each region of `partition`), `summarize(partition,val)` (summing `val` in each region of `partition`), and `localDensity` (estimating local density of devices). There, the advanced type system of Scala (including *generics*, *implicits*, *witnesses* and *defaults*) is of great help in making such APIs expressive, reusable and modular.

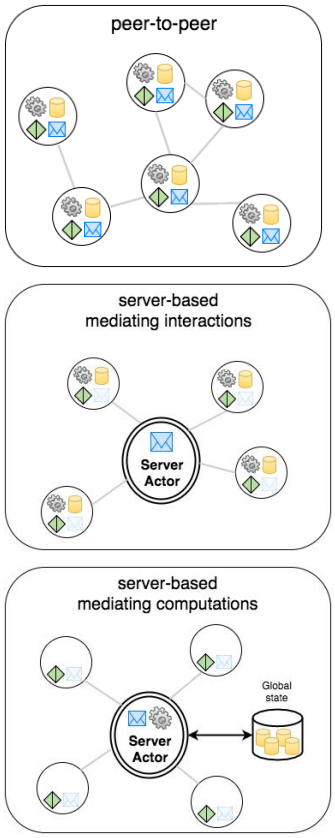

Figure 3: Three instances of `scafi` platform: (top) purely peer-to-peer, (middle) server-based mediating communications, and (bottom) server-based mediating computations. Transparent actors denote simplified versions of the basic actor.

Such APIs can, in turn, be used to implement application-specific aggregate behaviours. A reference example, following [1], is that of crowd engineering, in which functions of crowd detection, navigation, and dispersal can be developed. For instance, estimation of dangerous density according to [5], can be defined as follows (note the global character of these kinds of specification, which under-the-hood just drive local computations and the content of messages sent to neighbours):

```
def dangerousDensity(): Boolean = {
  val partition = randomRegions(60) // region size
  // danger = high local density and crowded region
  average(partition, densityEst()) > 2.17
    && summarize(partition, 1) > 300
}
```

Platforms

`scafi` is intended to provide flexibility in the execution and deployment aspects of aggregate systems. In particular, it decouples the specification of the aggregate program, the configuration of the system, and the choice of the platform incarnation; it does so by keeping a neat distinction between the physical computational network and the logical aggregate programming abstractions. In general, the construction of an aggregate system involves the specification of four pieces of information: *Choice or definition of a `scafi` incarnation*, which represents the selection of the desired platform features; *Specification of the aggregate program*, i.e., the system-wide computation; *System and platform configuration*, involving aspects such as deployment and instrumentation; and *System or node setup and launch*, including device bootstrap, definition of neighbouring relation, and attachment of sensors/actuators.

P2P actor-based

The most natural system architecture reflects the logical, spatial model of aggregate programming, where devices interact with each other on a local basis—in a fully decentralised manner.

In general, a device actor can be thought of as composed of multiple (child or attached) actors, each one assigned with a single, specific responsibility. Figure 2 depicts a complete (i.e., fully operational) device. Sensor and actuator actors handle interaction with the environment. If the device computes on-site, it has a computational actor, which is triggered by clock signals as sent by an internal or external entity. Such a computation actor queries the state actor for inputs—which include the result of the previous computation, the local sensor values and the exports of neighbour devices. Finally, the communication actor is responsible for getting exports from the neighbourhood and propagating the result of each computation round nearby.

In the peer-to-peer platform style (Figure 3, top), concretised by the `BasicActorP2P` incarnation, the system is a network of devices (nodes) represented by actors that, at each scheduled round of execution, compute the aggregate program and broadcast the result message to their neighbourhood. At this level, we abstract from the way the neighbourhood set is discovered: for example, it may be given at configuration time or provided by a neighbouring sensor.

Figure 4 shows how a programmer can easily start a node with a default configuration.

Actor mediating interactions

In some cases, it may be useful to move some of the duties of the devices to a central entity: an actor that can

```
// STEP 1: CHOOSE INCARNATION
import scafi.incarnations.{
     BasicActorP2P => Platform }
import Platform.{AggregateProgram,
  Settings, PlatformConfigurator}

// STEP 2: DEFINE AGGREGATE PROGRAM
class Program extends AggregateProgram
              with CrowdSensingAPI {
  // Specify aggregate computation
  def main() = dangerousDensity()
}

// STEP 3: PLATFORM SETUP
val settings = Settings()
val platform = PlatformConfigurator.
     setupPlatform(settings)

// STEP 4: NODE SETUP
val sys = platform.
     newAggregateApplication()
val dm = sys.newDevice(
  id = Utils.newId(),
  program = Program,
  neighbours = Utils.discoverNbrs())
```

Figure 4: Setup of a node in the P2P platform style.

```
import scafi.incarnations.{
  BasicActorServerBased => Platform}
... // STEP 1,2,3,4 as of P2P version
dm.addSensorValue(
  name = Utils.LocationSensorName,
  provider = ()=>Utils.getLocation())
```

Figure 5: Setup of a node in the platform style based on a mediator of interactions. The code is mostly the same as in the P2P case. The selection of a different `scafi` incarnation gives a new semantics to all the above method calls. In addition, we need to configure a location sensor providing device position to be sent to the server.

provide system-wide services and encapsulate environmental features.

As a first example, a server can mediate all device communications by keeping a representation of the space in which they are situated (which may be purely logical or a representation of the physical space and situation), and hence use a configurable distance metric to reify an application-specific notion of neighbourhood for each device. In the crowd steering settings, one such server would easily overcome the difficulties of smartphone local interactions (e.g., via Bluetooth).

Figure 3 (middle) illustrates one such platform style that is based on a central actor working as a mediator for device-to-device communications. This new platform is essentially obtained by a simple reallocation of responsibility, where the communication burden (and knowledge of neighbours) is moved from each device's communication actor to a single server's actor, receiving local computation results and sending the neighbourhood state.

Actor mediating computations

In another scenario we may move the aggregate computation from devices to the central server: the devices collapse to system sensors and actuators, essentially becoming environmental contexts upon which the aggregate system can perceive and act. The situation is represented by Figure 3 (bottom). Computation, state management, and neighbourhood communication responsibilities move from device actors to the central actor, which uses a (persistent or in-memory) database to store the global field.

This approach could provide a number of benefits: devices could be unaware of the actual aggregate program to run (which can then be modified on-the-fly in the server), and global optimisation techniques could be adopted to avoid computing all rounds in all devices.

Mixing actor mediating/computing

We have seen so far that a plausible execution architecture for aggregate systems can be based on a centralised entity which can, for example, be implemented as an actor. This server can be in charge of locality-based information propagation or computation. A possible next step is to envision a server which dynamically switches between merely mediating communications or computing aggregate programs.

The key insight of this paper lies exactly in the independence of an aggregate computation from the underlying execution strategy. In fact, thanks to our "pulverisation" semantics, an aggregate computation can be ultimately performed at the device site or by a computing entity that is able to correlate global and local information.

There may be practical reasons to opt for centralised execution platforms—e.g., for easier maintenance, to enforce security policies, or because broadcasts to neighbourhoods are not supported at the infrastructure-level. Secondly, the generality and abstractness of aggregate computing can provide greater flexibility with respect to *how* the ensemble ultimately carries out computations. This means that (i) the platform can make the best use out of the computational and networking resources at hand, and (ii) it can opportunistically adapt the execution strategy to changes in the available environment or computational infrastructure. It is possible to reason in terms of movable or fluent responsibilities, in the sense that certain operations can "flow" from devices to computing servers, or viceversa—dynamically.

Aggregate Computing in the Cloud
The introduction of central servers seems to contradict the original purpose of the aggregate computing approach, which fits fully decentralised distributed computing scenarios. However, handling large numbers of devices is possible using cloud-oriented approaches.

Cloud computing is a well-established model and technology supporting scalability and elasticity through on-demand provisioning of IT resources—which are typically virtualised. Since it represents a further opportunity for building scalable systems, it is reasonable to think of an alternate execution strategy for aggregate systems where computations are carried out in the cloud.

A main strategy for a cloud-based execution platform consists in *storing the whole computational field as a big data*, with aggregate computation structured as a myriad of stateless computing services concurrently working on a big shared database.

The global aggregate computation might even be executed partially "on ground", e.g., as advocated in edge- and fog-computing initiatives [2], and may "flow" up and down depending upon context and contingencies—energetic issues, presence of congestions, unexpected storage requirements, changes in wireless availability, and so on.

Conclusions and Future work

In the future, the goal of this research is to evolve `scafi` into a programming model and framework dynamically supporting scalable computations in cluster- and cloud-based systems. In a sense, we would like to achieve the same benefits of frameworks such as Hadoop and Spark, where the Aggregate Computing paradigm plays the role of the MapReduce programming model. However, while MapReduce is limited to data processing, aggregate programming supports distributed and situated computations—basically what the IoT scenario demands.

REFERENCES

1. Jacob Beal, Danilo Pianini, and Mirko Viroli. 2015. Aggregate Programming for the Internet of Things. *IEEE Computer* 48, 9 (2015).
2. Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. 2014. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*. Springer, 169–186.
3. Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. 2015. Code Mobility Meets Self-organisation: A Higher-Order Calculus of Computational Fields. LNCS, Vol. 9039. Springer, 113–128.
4. Alois Ferscha, Paul Lukowicz, and Franco Zambonelli. 2015. Collective adaptation in very large scale ubicomp: towards a superorganism of wearables. In *Proceedings of the 2015 ACM UbiComp/ISWC Adjunct 2015*. ACM, 881–884.
5. John Fruin. 1971. *Pedestrian and Planning Design*. Metropolitan Association of Urban Designers and Environmental Planners.
6. Marco Mamei and Franco Zambonelli. 2009. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. on Soft. Eng. and Methodologies* 18, 4 (2009), 1–56.
7. Mirko Viroli, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. 2015. Efficient Engineering of Complex Self-Organising Systems by Self-Stabilising Fields. In *IEEE SASO 2015*. 81–90.