Towards Attack-Resistant Aggregate Computing Using Trust Mechanisms

Roberto Casadei[a], Alessandro Aldini[b], Mirko Viroli[a]

[a]*Alma Mater Studiorum—Università di Bologna, Italy*
[b]*Università di Urbino Carlo Bo, Italy*

Abstract

Recent trends such as the Internet of Things and pervasive computing demand for novel engineering approaches able to support the specification and scalable runtime execution of adaptive behaviour for large collections of interacting devices. Aggregate Computing is one such approach, formally founded in the field calculus, which enables programming of device aggregates by a global stance, through functional composition of self-organisation patterns that is turned automatically into repetitive local computations and gossip-like interactions. However, the logically decentralised and open nature of such algorithms and systems presumes a fundamental cooperation of the devices involved: an error in a device or a focused attack may significantly compromise the computation outcome and hence the algorithms built on top. For this reason, in this paper, we move the first steps towards attack-resistant aggregate computations. We propose *trust* as a framework to detect, ponder or isolate voluntary/involuntary misbehaviours, with the goal of mitigating the influence on the overall computation. On top of this, we consider recommendations in order to provide more reactivity and stability through the sharing of individual perceptions. To better understand the fragility of aggregate systems in face of attacks and investigate the extent of the mitigation afforded by the adoption of trust mechanisms, we consider the paradigmatic case of the *gradient* algorithm. Experiments are carried out to analyse the sensitivity of the adopted trust framework to malevolent actions and to study the impact of different factors on the error committed by trust-based gradients under attack. Finally, in a case study of the *spatial channel* algorithm, it is shown how the protection afforded by attack-resistant gradients can be effectively propagated to higher-level building blocks.

Keywords: Aggregate Computing · Computational fields · Collaborative P2P systems · Security · Safety · Trust.

1. Introduction

The last decades have been feeding a process where large numbers of interconnected computing devices get densely deployed in our living and working environments. Such technological and social movements are implying a future of increasing pervasiveness and interconnection, where dense networks of computer-like nodes are overlaid on and tightly interact with our physical world and humans in it. Exploiting one such computational fabric is appealing but it does challenge current methods and tools in a paradigmatic way: the large-scale, situated and complex nature of this kind of systems makes open-loop approaches infeasible and pushes forward the need to embrace self-* properties, but hence a whole set of new challenges arises.

The field of collective adaptive systems (CASs) [1] is devoted to the study of systems where large groups of entities jointly seek to reach their goals in a dynamic environment [2]. The main issues in this context include *(i)* how to provide an effective specification of self-organising and goal-oriented behaviour and *(ii)* how this can turn into efficient, resilient and distributed execution. Moreover, from the engineering side, trade-offs have to be carefully balanced to limit usage of resources and still provide the required quality of service. Given this complex setting and the impracticality of in-field tests, it is crucial to be able to count on ways to obtain certain guarantees on the correctness of implementations; for this purpose, both formal methods and simulations are invaluable.

Aggregate Computing [3] is one promising approach for the engineering of (possibly large-scale) distributed systems that need to resiliently adapt to local, environmental conditions. The idea is to directly express, by a single global program, the behaviour of an *aggregate* of devices, all of which interpret that program and correspondingly execute local actions encompassing continuous sense of the environment, computation of local data, and their sharing with

neighbours. This approach takes an abstract, global stance in which one programs the desired collective behaviour, through a composition of self-organising coordination patterns [4], and lets the platform deal with the proper unfolding of the computation at the micro-level in a complex set of repetitive, weaved interactions and calculations. The field calculus [5, 6, 7], which builds on the idea of computational fields (spatially-distributed data structures), provides the formal underpinnings of Aggregate Computing and gives a concrete shape to the approach: in this framework, programs at the aggregate level are represented as functional compositions of dynamic fields that map devices to computational values in space-time. In practice, an aggregate system consists of a collection of networked devices where each device computes the same aggregate program and interacts with a subset of other devices known as its neighbourhood. That is, computation unwinds in a logically decentralised way based on locally sensed information and data received through peer-to-peer, gossip-like communications.

Among the various challenges behind the design and development of successful CASs, trust represents a fundamental aspect in a setting in which the rapid and continuous exchange and propagation of information is key. The need for cooperation is typically accompanied by the growth of potential (insider) threats, which may depend on selfish or malicious behaviours of nodes, either in isolation or in collusion. From the viewpoint of a node issuing a request to another node, which may refer to the communication of a simple detected value or the delivery of a complex paid service, trust can be defined as the belief perceived by the former node about the capability, intention, honesty, and reliability of the latter node in satisfying the request. The estimation of such a belief perception through automatic mechanisms supporting the decision-making processes improves the reliability of CASs and, therefore, it is quite natural to investigate their application also to the Aggregate Computing framework. However, implementing a trust model respecting the principles of Aggregate Computing is particularly challenging in a distributed setting in which the promptness and success of self-adaptation strongly depends on the accuracy and reliability of (partial) information exchanged among interacting devices.

Starting with these considerations, we propose the combination of trust and Aggregate Computing, with the goal of making collective, cooperative computations more resilient to voluntary or involuntary misbehaviours. The results of the experimental analysis of the proposed approach are promising: the use of trust mechanisms can effectively reduce or nullify the error caused by subversive behaviours, thus contributing to alleviate the fragility inherent to open, cooperative computations.

This paper is organised as follows. In Section 2, we recall the paradigm of Aggregate Computing, its formal instantiation in the field calculus framework, and describe the basics of programming with computational fields—using ScaFi as reference implementation. Here, we also introduce the "collective algorithms" considered in the evaluation part of the paper—namely, the self-healing, self-organising gradient and channel algorithms. In Section 3, we provide an overview of the security issues in Aggregate Computing, motivating this study and its focus. Then, in Section 4, we discuss and refactor the applicability of classical trust management techniques to the Aggregate Computing setting. Section 5, hence, describes a trust-based implementation of the gradient algorithm. Then, in Section 6, we present a comprehensive empirical study of the effectiveness of trust fields in mitigating or avoiding at all the consequences of (deliberate or not) misconducts of nodes in a gradient computation. An empirical study of the benefits of using trust-based gradients in a more complex computation—i.e., the channel— is considered in Section 7; this proves the effectiveness of the approach in aggregate algorithmic compositions that include attack-resistant building blocks. Finally, comparison with related work and future perspectives are discussed in Section 8.

Most specifically, this paper extends the work in [8] with: sensitivity analysis on the main factors affecting the trust algorithm (Sections 6.2, 6.3); description, implementation and analysis of recommendations on top of the plain trust algorithm (Sections 4, 5, 6.3); refactoring and description of the source code for the trust-based gradient algorithms (Section 5); empirical study of the response of the trust algorithms against mobile attackers (Section 6.3); and empirical study of the benefits of selecting a trust-based gradient algorithm in a more complex computation, namely the *self-organising channel* (Section 7).

2. Aggregate Computing

In what follows, we provide a general description of the Aggregate Computing paradigm, then we introduce the formal framework that makes it operational, and finally show a progression of examples covering the basic elements of the framework up to the components considered in the evaluation sections.

2.1. The Aggregate Computing paradigm

Aggregate Computing [3] is a paradigm and an engineered stack of software layers aimed at supporting the development of a large class of distributed systems, such as those conceived in IoT and pervasive computing scenarios. It addresses in a principled way many challenges found in distributed systems (communication, concurrency, fault-tolerance), situated systems (time-, space-, and resource-aware coordination), and CASs (aggregate behaviour, self-* properties).

As suggested by the term itself, Aggregate Computing adopts a global stance to distributed computation and coordination design: the target of programming is not the individual computing node (generally referred to as a *device*), but rather an entire *aggregate* of networked devices, which is seen as a conceptually single (yet distributed) computational machine—possibly situated in some physical environment and evolving dynamically as devices move or fail or network partitions occur. In practice, the approach works as follows:

i) the application development team designs and implements an *aggregate program* (or service) expressing the intended collective behaviour in terms of a model of the environment (formalised through the required sensing capabilities of devices);
ii) the devops team deploys the *aggregate application* to the set of smart objects of an IoT environment (*aggregate system*), either directly to all of them, or by injecting the code into one device and let it dynamically diffuse [6];
iii) execution leverages the Aggregate Computing middleware to: schedule executions of the aggregate program, take care of communication between devices across the network, create the computational context local to each device, and provide access to the Aggregate Computing virtual machine for interpreting a given aggregate program contextually to the running device through proper *global-to-local mapping*.

Like for traditional paradigms, application development may benefit from both high-level aggregate functionality (e.g., for drone flocking or crowd management) and lower-level, generic constructs (e.g., for propagation and collection of information across peer-to-peer networks). Therefore, crucially for scaling with complexity, the Aggregate Programming model is compositional: aggregate behaviour can be designed by properly arranging building blocks of predictable compositional semantics together. To give an intuition of what this means, a simple logic for distributed sensing can be conceived as the following global specification, continuously executed by a (possibly large-scale) network of situated devices: *(i)* split the space into partitions of a given average extension, *(ii)* elect a leader in each partition, *(iii)* collect the sensing estimates from the devices of each partition to the corresponding leader, and finally *(iv)* propagate in turn the mean value computed by each leader within the corresponding partition. Ideally, each step should be as simple as calling one library function.

Another key characteristic of Aggregate Computing is *abstraction*, which is instrumental for achieving declarativeness in code and driving run-time adaptation. More specifically, the logic and the correctness of an aggregate program may be quite independent from a set of conditions such as the topology of the network and the density of the devices situated in some region. In addition, this generality provides some valuable flexibility at the platform-side: though the approach encourages a fully decentralised mindset, there is large freedom with respect to which concrete execution strategy can operate an aggregate system [9]—ranging from completely peer-to-peer to centralised (server- or cloud-based), up to hybrid and adaptive ones [10] (e.g., according to the available infrastructure).

2.2. Computational fields

The formal underpinning of the Aggregate Computing framework is given by the *computational field* notion (or *field* for short) and the corresponding *field calculus* [5, 11, 7]. A computational field is a data abstraction mapping devices in a system (which form the *domain* of the field) to computational values over time; i.e., it is a distributed data structure whose values are given by the outputs yielded by the devices participating to the computation as they repeatedly run their program. If, as it is often the case, devices are situated in some space, fields can also be thought of as functions from space-time points to the values produced by the devices at those locations. The time-wise nature of fields is what accounts for dynamics, whereas the spatial dimension provides a foundation for context and interaction. Indeed, each aggregate application defines a notion of *neighbourhood* that determines whether two devices are allowed to communicate; natural neighbouring relationships are those that merely result from physical/networking connectivity and spatial locality, but also logical ones are possible in principle (e.g., social relationships). Finally, notice that a field can be interpreted both "punctually", or device-wise (micro-view), and "globally", or system-wise (macro-view), opening the way for bridging individual and aggregate behaviour.

The field calculus is the principal formal framework for working with fields. It defines a minimal set of constructs for transforming and combining fields together, and provides a proper basis for formal analysis of *field expressions*, formed by *field values* (or *fields* for short[1]) and *operators* on fields, and reasoning about *field computations* obtained as a result of their evaluation. The key feature of this calculus is compositionality, by which complex field computations can be described in terms of simpler ones, thus enabling the development of libraries and layers of aggregate functionality [3]. As we shall see, field computations are run in a repeated fashion by a collective of devices, and typically leverage sensing capabilities to provide adaptivity with respect to environmental change. In such a setting, an important property is *self-stabilisation* [12]—holding for a computation which, independently of the initial state considered, eventually reaches a correct final state if environmental change ceases. Notably, a subset of the field calculus [13] has been identified for which it is proved that any generated field expression leads to a self-stabilising computation.

In what follows, the execution model and the constructs of the field calculus are described with reference to the ScaFi implementation. ScaFi [14] (Scala Fields) is a JVM-based open-source framework[2] for Aggregate Computing. It provides a Scala-internal Domain-Specific Language (DSL) and a corresponding interpreter for expressing and running aggregate computations according to the field calculus, as well as actor-based platform support for building distributed aggregate systems.

2.3. On deployment and execution of aggregate systems

An aggregate program describes the global computation to be carried out by an aggregate system, and also works as the unit of application deployment. It consists of a main field expression representing the entry point of the program, possibly accompanied by a set of functions for modularity. In ScaFi, it can be defined as follows:

```
class MyAggregateProgram extends AggregateProgram with Lib {
  override type MainResult = Double // Set the type of the computation result
  override def main = f+g           // g is provided by Lib

  def f: Double = 7.7               // (constant) function returning a field
}

trait Lib {
  self: AggregateProgram => // Self-type: means Lib must be injected into an AggregateProgram
                            //            (and thus can use functionality of AggregatePrograms).
  def g: Double = 3.3
}
```

This program simply returns field `11.0`; i.e., all the devices continuously compute `11.0`. The main expression becomes a method `main` of a class extending `AggregateProgram`, whose body can call local methods introduced for modularity, constructs of field calculus inherited from class `AggregateProgram`, or functions defined in other libraries mixed-in with clause `with`—other details about the Scala syntax and field computations are covered in Section 2.4.

Aggregate systems can be deployed according to different architectures [9]. In a typical peer-to-peer scenario, an aggregate program is deployed to a set of devices geared with the Aggregate Computing middleware and is locally executed according to the operational semantics of the field calculus. Basically, an Aggregate Computing middleware (or *platform*) provides: *(i) field calculus virtual machine*, for executing field computations against the device's local computational context; and *(ii) middleware services*, to manage the computational context, the lifecycle of devices, the communication among devices (according to a neighbouring notion), while also dealing with distribution issues.

ScaFi provides an actor middleware based on Akka [15], which is covered in detail in [16]. The idea is that a logical device of an aggregate system becomes an actor that interacts via message-passing with other components

[1] Notice that, as common abuse of terminology, we use term "field" to refer to both the type and its values—analogously, e.g., when one can say "1 is an integer (value)" or "1 has type integer".

[2] `https://github.com/scafi/scafi`

of the middleware, other actors representing neighbour devices, as well as with other message-oriented or event-driven entities of the system in which it is deployed. By partitioning an aggregate application into a logical system of actors [9], it is possible to choose among different deployment strategies (i.e., the ways in which such actors are mapped to physical or virtual machines) and even perform dynamic adaptations at runtime, through proper migration of actors.

Ultimately, the overall behaviour of an aggregate system is a result of both the aggregate program specification and the details of the runtime execution. Indeed, a system enacting field computations has to work in accordance with an abstract execution model where devices iteratively run their program and communicate with one another in order to cooperate in building the local contexts and driving micro-level activity. Each device repeatedly runs the same aggregate program (which might branch differently throughout the nodes, though), alternating sleep periods so that computation locally proceeds at discrete rounds of execution. From a global point of view, computation is usually fair and partially synchronous, though these assumptions may change or be relaxed on a case-specific basis. At any round, a device *(i)* builds its up-to-date local context by collecting previous computation state, sensor values, and messages received from other devices, *(ii)* runs the aggregate program, which produces both a result value as well as a description of the just-performed computation that is known as the *export*, *(iii)* shares its export, e.g., through a broadcast, to its neighbour devices, as defined by an application-specific notion of neighbourhood, which typically relies on physical distance, and finally *(iv)* executes the actuators as specified by the program.

The export is a tree-like descriptor of an aggregate computation that is communicated and used by devices to safely interact with one another. In fact, in this framework, a device is allowed to interact – at a given point of the computation, namely at a given point in the export tree that is currently being built – only with the set of its *aligned* neighbours. This process, known as *alignment* [17], is a key mechanism in that it supports consistent execution of aggregate computations and also provides the means for splitting computations (fields) into completely separated branches (field partitions).

Notice that the steps of a computation round need not to be strictly sequential and may be triggered at different frequencies. Also, the frequency at which rounds are executed can vary throughout the lifetime of a single device and from device to device; for instance, a device may slow down or increase the rate of its operation based on battery levels or speed of environmental change. Such independence from low-level execution details paves the way to optimisations and operational flexibility [9].

2.4. ScaFi *and the field calculus constructs*

ScaFi provides access to aggregate programming features together with the type system of Scala and all the amenities that can be found in a mainstream programming language supporting imperative, object-oriented and functional paradigms. Most specifically, field expressions can be given as regular Scala expressions: there are no explicit "first-class" fields in ScaFi in that, e.g., any `Double` expression is naturally interpreted as denoting a field of double values, and similarly for any other Scala type. Accordingly, the core constructs for working with fields are reified as plain, generic methods, defined in the following Scala interface.

```
trait Constructs {
  // Key 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)

  // Contextual, but foundational
  def mid(): ID
  def sense[A](name: LSNS): A
  def nbrvar[A](name: NSNS): A
}
```

In Scala, methods are introduced with the `def` keyword, have their return type specified at the end of the method signature, can be generic (type parameters are specified within square brackets) and can be defined in curried form by providing multiple parameter lists; when invoked, 1-element parameter lists can be equivalently specified with round

or curly brackets (in the latter case, it visually emulates block-like structures, which is nice for DSLs), while with 0-element parameter lists one can even avoid the empty list of parameters "`()`"; syntax `(T1,T2)` denotes a 2-element tuple type (and there is coherent syntactic sugar for tuple values); syntax `A=>R` denotes a function type (and there is similar syntactic sugar for lambdas); and syntax `=>R` denotes a call-by-name parameter that is passed unevaluated to the method body (as a thunk) and in there gets (re-)evaluated at each use (basically it is a syntactic shorthand over nullary functions).

When discussing field constructs, as well as when reading and writing aggregate programs, two complementary perspectives can be adopted: the local, device-centric one (which relates to the operational semantics) and the global, field-centric one. For example, literal `5` can be thought of as either a single number calculated by a specific device or as a uniform, constant field of fives. Intuitively, a field is *uniform* if it does not change across devices or, equivalently, across space (under the assumption of taking snapshots at synchronous rounds), whereas it is *constant* if it does not change across time. An example of non-uniform (but constant) field is given by:

```
mid // same as: mid()
```

where `mid` is the 0-ary function that returns the identifier of the running device (local interpretation) or, equivalently, the field of device identifiers (global interpretation). Non-constant fields, instead, may result from stateful computations or sensor readings. Traditional local types and operators are transparently lifted to field types and field operators. For instance, expression `"hello␣" + "scafi"` can be interpreted locally in the conventional way or globally as a field expression combining the two field values `"hello␣"` and `"scafi"` with the field operator `+`, which behaves as the local counterpart (i.e., concatenates strings) but does so in a global-wise fashion, yielding a field of `"hello␣scafi"` strings.

Construct `rep(init)(fun)` yields an `init` field that evolves over time according to the given state transformation function `fun`. Note that such a progressive transformation is not atomic nor necessarily homogeneous, as it depends on when the devices that make up the field actually fire. For example,

```
rep(0)(x => x + 1)
```

produces a non-constant field counting the number of rounds performed (point-wise, i.e., in each device).

In the field calculus, communication is achieved through *neighbouring values*—sometimes called neighbouring fields. A neighbouring value can be thought of as a local value made of a map associating to each neighbour some value it previously shared with the current device; in other words, one such map represents the result of "observing" the neighbourhood, e.g., to retrieve values sensed by all neighbours, an estimation of distances from neighbours, and so on. Notice that a device's neighbourhood includes the device itself (reflexivity property). Now, construct `foldhood(init)(acc)(fexpr)` is what allows you to reduce, in each device, the neighbouring value `fexpr` (using `init` as the identity field for the binary operation `acc`) into an atomic value—seen at the global level, hence, `foldhood` takes a field of neighbouring values and reduces it to a field of local values. More precisely, it folds over the set of *aligned* neighbours, evaluating `fexpr` each time with respect to a different neighbour. Notice that `fexpr` is a by-name argument and hence it is passed unevaluated to the function, to be evaluated on a per-use basis. `nbr(nexpr)` is the basic operator that supports communication of information among devices[3], i.e., the creation of neighbouring values. The semantics of `nbr` depends on the neighbouring context in a certain `foldhood` iteration step: for the running device, `nexpr` is evaluated and the corresponding result is returned (as well as written in the export so that such value can be broadcast in turn); for the other neighbours, the corresponding most recent outcome of `nexpr` available in the running device is retrieved. For example,

```
foldhood(false)(_ || _)(nbr { sense[Boolean]("alarm") })
```

denotes the field of all the devices that sensed an alarm or that are nearby a device that did. In general, `sense[T]("name")` represents the field of readings of a value of type `T` from a sensor called `"name"` (which is assumed to be in place); punctually, it corresponds to a query to a local sensor (it is a matter of the platform to bridge these logical sensors to physical ones, in case). Moreover, there is a special kind of sensors (known as *neighbouring*

[3]Note that the actual communication between devices is matter of the platform and is usually performed through export broadcasting (and not during program execution).

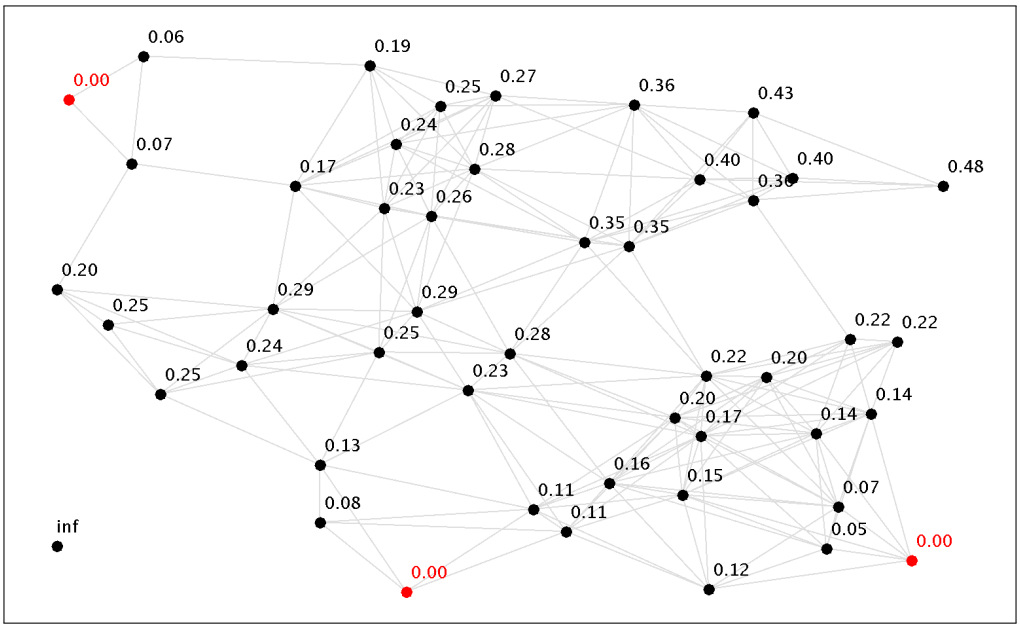


Figure 1: Snapshot of a stabilised gradient field from a simulation in ScaFi. Devices are represented by bold dots; the red ones denote the sources from which the gradient is computed. The grey lines represent neighbouring links: connected nodes exchange their exports. The nodes are labelled with the string representation of the result value of the local computation.

sensors) which are read through `nbrvar`. Similarly to `nbr`, `nbrvar` yields a neighbouring value and so it needs to be used in a `foldhood` expression. Neighbouring sensors are of use for mapping each neighbour set to platform-level readings; the way such sensors are implemented is a platform issue. A common one is `nbrRange`, that yields the field of neighbouring values representing estimated distances of neighbours from current device:

```
def nbrRange = nbrvar[Double]("NBR_RANGE")
```

So, for instance, when `nbrRange` is evaluated in a `foldhood` iteration, it will return the distance corresponding to the aligned neighbour considered in each iteration step (which will be `0.0` for the running device). Also, notice how plain Scala functions can be used to encapsulate field computations for reuse.

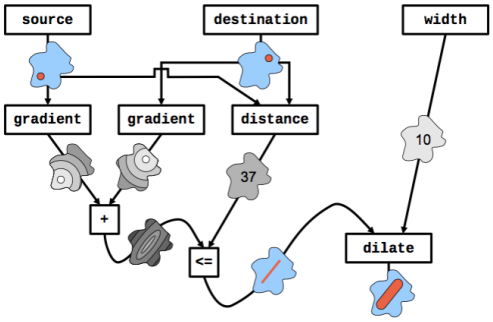
The last foundational construct is `branch(cond)(e1)(e2)`, which splits the computation into two subcomputations `e1` and `e2` according to a boolean field `cond`. This effectively splits the domain into two subdomains: the devices for which `cond` is true, and those for which `cond` is false; the former group will continue the computation with `e1`, while the latter with `e2`. Notice that the parameters of `branch` are by-name arguments, since they need to be evaluated lazily.

```
branch(sense[Boolean]("danger")){
  // computation to be run when a danger is perceived
}{
  // alternate computation to be run
}
```

2.4.1. Example: the gradient field

Now we show a simple example that will also be used in the security-related experiments of Section 6. It is known as the *gradient* [19] and represents the field of minimum distances from source nodes (Figure 1). In ScaFi, it can be implemented as follows:

```
def gradient(source: Boolean): Double =
```



(a) The channel as a composition of field computations. Notice the foundational role played by gradient fields—picture taken from [18].

```
def channel(src: Boolean, dest: Boolean,
            width: Double, g: Boolean => Double):
  Boolean = g(src) + g(dest) <=
              distBetween(src, dest, g) + width

def distBetween(src: Boolean, dest: Boolean,
                g: Boolean => Double): Double
  = broadcast(src, g(dest))

def broadcast[V:Bounded](src: Boolean, field: V): V
  = G[V](src, field, v => v, nbrRange)
```

(b) Implementation of the channel algorithm. Notice the coherence of the source code for function `gradient` with the pictorial representation.

Figure 2: The channel building block.

```
rep(Double.PositiveInfinity){ distance =>
  mux(source) { 0.0 } {
    foldhoodPlus(Double.PositiveInfinity)(Math.min)(nbr{distance}+nbrRange)
  }
}
```

Function `mux(cond)(v1)(v2)` is a purely functional multiplexer for selecting between two values `v1` and `v2` depending on a boolean value `cond`. Where the `source` field is true, the gradient is zero; otherwise, the new gradient estimate is built by taking the minimum value among the neighbour estimates augmented by the corresponding node-to-node distance. Notice that we must use `mux` instead of `branch` in order to allow source nodes to share their values—to do so, they need to evaluate the `nbr`s in the else-branch of the `mux`. The outer `rep` is necessary to keep track of the estimated distance from one round to the next. `foldhoodPlus` is a (derived) variant of `foldhood`

```
def foldhoodPlus[A](init: => A)(aggr: (A, A) => A)(expr: => A): A =
  foldhood(init)(aggr)(mux(mid==nbr(mid)){ init }{ expr })
```

that maps the running device itself with the "initial" value of the folding (in this case, `Double.PositiveInfinity`): it is necessary to prevent a node from sticking to its value when gradient values arise. Notice that this gradient algorithm is *self-healing*, in that it reacts to perturbations (e.g., as triggered by mobility or change of sources) by starting a process that steers the field towards the correct shape, and it is self-stabilising as well, since it eventually reaches a stable state once environmental inputs cease.

The gradient is a fundamental building block for self-organising computations [4]: it is internally used by several higher-level components that realise functionalities such as bounded information broadcast, dynamic network partitioning, and distributed sensing. As such, improvements to its performance and effectiveness (e.g., reactivity, stability or resilience to attacks) can have a huge positive impact on applications [19].

2.4.2. Beyond the gradient: the channel

A simple but significant example to show how the gradient can be used to build more complex computations is the *channel* structure [18]: an adaptive field of booleans denoting the path from a source area to a target area. The logic behind the channel design and the corresponding implementation are shown in Figure 2. The channel is parameterised with a shortest path estimation function `g` (lowercase), which may be the `gradient` function shown before or a different gradient implementation (e.g., that adopts trust mechanisms as covered in the case study of Section 7). For the sake of completeness, notice that the `broadcast` function, which is used to globally propagate the source-to-target distance from the source outwards, makes use of `G` (uppercase)—a generic coordination operator allowing the propagation of information along a gradient (which then becomes a specific case of `G` application where distances are accumulated).

3. Aggregate Computing and Security

Computer security is widely recognised as a fundamental aspect in the engineering of computer-based systems. This section overviews the particular relation between Aggregate Computing and computer security, which is a significant problem due to the suitability of the approach to safety-critical applications such as tactical networks or crowd management; for instance, on the latter, case studies have been considered [3, 20] based on urban scenarios (e.g., during a mass public event such as a marathon) where the people's smartphones collectively run aggregate services for crowd warning and crowd-aware navigation.

Basically, Aggregate Computing systems are susceptible to the security threats that naturally arise from distribution and situatedness; additionally, openness can make participant nodes – over which little control and knowledge is available – leave and join the computation at any time.

The attack surface of aggregate systems may be very large. In practice, attacks may target infrastructure, physical devices, the Aggregate Computing platform, and/or the application logic. At the infrastructure level, the physical environment may be manipulated or the communication network interrupted. At the level of the physical device, the hardware as well as sensors and actuators may be impaired or hijacked. At the platform level, databases may be sabotaged and crafted messages (data payloads or commands) may be issued.

The ability of Aggregate Computing to abstract over the particular system execution strategy [9] can provide some resistance to certain types of attacks or give design choices to be further evaluated by a security perspective. For instance, infrastructural malfunctioning could in principle be tackled by adapting interaction and computation loci according to the available communication media and endpoints. Moreover, by logical design, Aggregate Computing fosters decentralised systems and ad-hoc networking where computation is physically distributed and peer-to-peer interactions happen according to physical proximity, hence avoiding single points of failure such as central servers or cloud endpoints.

By a functional point of view, the decentralised nature as well as the peculiar characteristics of adaptivity of the approach make aggregate applications resilient to intermittent or prolonged failure of some nodes—especially for self-stabilising computations like the gradient [12]. However, the actual impact of node malfunctioning depends on many factors, such as the topology of the network or how dense it is (i.e., the extent of "spatial redundancy"): network partitions may prevent information to reach significant regions of a collective, and sparseness increases the relevance of individuals. Also, the role that a node plays in the application is crucial: for example, behaviours building on the gradient algorithm (Section 2.4.1) may be significantly altered if the source nodes fail and the other nodes at some point get rid of the past export, though not necessarily in short time, as it also depends on the frequency of round execution, the amount of time that neighbour exports are retained (which depends on the platform configuration), and the transitory phase of the particular gradient algorithm in action.

In this paper, we are most interested in attacks at the application-level, i.e., in attacks that work by producing factitious export messages. Basically, we assume that nodes are untrusted and, potentially, can create and inject fake messages that can be received and used by their neighbourhood [21]. These fake messages can be of two types: malformed or well-formed. Recall that, in our framework, the aggregate virtual machine ensures that interaction is only possible between aligned devices; for this reason, nodes emitting malformed messages cannot really participate into an aggregate application, for they would be automatically neglected.

More subtle is the case where well-formed (i.e., structurally-aligned) messages with malevolent payload are broadcasted. This basically accounts to injecting forged data at particular nodes in an export tree. A simple example is sharing negative values for the distance estimation in the gradient algorithm. In fact, the simple implementation of the gradient works by calculating the minimum of neighbours' estimates augmented by the respective distance, and it is easy to see how negative values would depress the field, misdirecting the original intention. In this particular case, the problem might be tackled by using more expressive types or by expressing constraints (e.g., via annotations) on the expected values, letting the platform deal with it; but in general, this would limit expressiveness and only partially solve the problem.

The misbehaviour enacted by a given malevolent entity can be more or less sophisticated. Its choices include *(i)* what kind of factitious data has to be generated, *(ii)* when, and *(iii)* who is the recipient. In fact, data can be randomly generated or suitably forged; also, an attacker may alternate good and bad communications (on-off misbehaving), and may limit bad communications to only a few targets (selective misbehaving) [22, 23]. Obviously, the complexity

and the potential of attacks can escalate when, instead of limiting ourselves to individual attackers, we also consider coordinated attacks by malevolent collectives.

At the heart of the problem is the fundamentally cooperative nature of aggregate applications: each device of an aggregate system, while preserving some autonomy with respect to mobility, sensing and actuation, is required to appropriately participate in the distributed aggregate computing process—which means executing the same program and not cheating. Of course, a proper security strategy would require the application of countermeasures across the whole Aggregate Computing stack—from humans and physical devices up to the programs. In the next sections, we consider how the problem can be tackled at the code-level and propose the use of trust mechanisms to deal with malicious payload.

4. Trust Framework and Its Application to Aggregate Computing

In several community-based domains, it is not possible nor convenient to maintain a trustworthiness infrastructure relying on centralised trusted third parties. To overcome such a limitation, trust relations are typically constructed on the base of direct observations and, possibly, recommendations gathered by interacting with the neighbourhood. For instance, in trustworthy crowdsourcing and sensor networks, a computational notion of trust derives from the exchange and aggregation of information disseminated by the participating nodes [24, 25, 26, 27, 28]. Once a trust metric is established, the usual trust-based decision-making policy consists of comparing the trust estimated by a node, called trustor, about the expected behaviour of another node, called trustee, and a trust threshold value *tth*, which may depend on several subjective factors, like, e.g., the initial willingness of the trustor to cooperate with the (possibly unknown) trustee.

In the computational trust literature, several metrics are based on a Bayesian approach. In essence, the trustor assumes that there exists an unknown parameter θ used to predict probabilistically the future good/bad behaviour of the trustee, and the related outcome is drawn independently for each interaction between them. In order to model uncertainty, θ is drawn by a given *prior* distribution, updated as new interactions between the parties occur. Among the various probability prior distributions proposed in the literature, the beta distribution received particular attention [29, 28, 24, 30]. Such a distribution is fed with two parameters, α and β, which count the number of positive and negative observations experienced by the trustor when interacting with the trustee, respectively. The evaluation of each observation depends on the context. As an example, in the setting of data relaying, a packet sent from the trustor that is forwarded (resp., discarded) by the trustee is considered as a positive (resp., negative) cooperation.

Then, trust is estimated as the statistical expectation $\mathtt{E}$ of a beta distribution Beta parameterised with respect to α and β, by assuming the initial scenario $\alpha = \beta = 0$, denoting absence of any prior interaction between the parties. Formally:

$$\mathtt{E}(\text{Beta}(\alpha + 1, \beta + 1)) = \frac{\alpha + 1}{\alpha + \beta + 2}.$$

Notice that the initial trust is equal to 0.5, which expresses a situation of total uncertainty about the expected behaviour of the trustee.

Different techniques based on such a Bayesian approach differ for the way in which *(i)* observations are weighted, e.g., depending on their age, and *(ii)* recommendations gathered via interactions with the neighbours are combined with the parameters discussed above.

The ageing mechanism can be implemented either by decreasing periodically the result of past observations by a weight w, or by assuming that only the last n observations contribute to the computation of trust. This kind of mechanism avoids the past behaviour to be too impairing over the current behaviour, thus mitigating the effect of on-off misbehaviours. Notice that we will consider the latter approach in the application to Aggregate Computing.

On the other hand, the recommended values received by a node from the neighbourhood, and related to the trust towards a specific trustee, are somehow combined to contribute to the computation of the subjective trust of the node towards the trustee. To avoid subtle colluding attacks, like, e.g., bad mouthing (fake negative recommendations about a honest node) and ballot stuffing (fake positive recommendations about a malicious node), the aggregation privileges direct observations with respect to evidences obtained from other nodes and weights such evidences by the trust towards the nodes providing them.

4.1. Application to Aggregate Computing

The Bayesian approach surveyed above can be applied also to the fully-distributed computational framework of Aggregate Computing. For the sake of simplicity, we consider the case in which nodes compute locally on the base of numerical values exchanged with the neighbours, as in the case, e.g., of the gradient field. In Aggregate Computing, the trustworthiness of the nodes depends on the quality of the information they share at each round. Hence, the two trust parameters α and β shall reflect such a relation. In order to estimate the quality of shared data, we first observe the following basic principle: if all the nodes are cooperative, the estimates of the gradient that every node receives from its neighbourhood in a round shall not be too much different from each other, up to certain fluctuations that may depend on several factors, like, e.g., the topology of the network, the location of the source node, the firing frequency of each node, and so on. Hence, if the value received from a node differs too much from the others, then such an observation is used to impair negatively the trust towards that node. In other words, unexpected perturbations of the gradient against the overall trend are considered as a potential attack to the system. On the other hand, a gradient estimate matching the general trend of the gradient field represents a good observation that can be used to affect positively the trust towards the node providing that value.

In order to implement this idea, we estimate trust by following the Bayesian approach in such a way that every gradient estimate received in a round is compared with the average of all the estimates received in that round. The detected difference is then used to evaluate the observation and update the parameters feeding the trust metric, by assuming that if the difference is evaluated positively (resp., negatively) then parameter α (resp., β) is increased. Since the mean square deviation, called s, represents a standard way to predict differences among values, we use it as the basis to compute the tolerance threshold, called *maxError*, for the evaluation of the difference above. In particular, we assume that the tolerance threshold is computed as a function dependent on s, whose definition represents a parameter of the trust system used to determine whether to deliver penalties, by increasing β, or rewards, by increasing α.

Once α and β are updated according to the policy above, they are possibly integrated with recommendations provided by the neighbourhood by using a mechanism inspired by [29, 24]. Then, the resulting pair of updated parameters feeds the Beta distribution that governs the computation of the trust metric, which is then compared against the trust threshold. Finally, only the gradient estimates received from trusted nodes are actually used to update the local estimate of the gradient.

Formally, each node i maintains locally the pair of parameters $(\alpha_{ij}, \beta_{ij})$ for each neighbour j. Their initial value is zero. At each round, node i performs the following operations:

1. Node i computes the mean $\bar{x}_i$ of the values x_{ij}, $1 \leq j \leq N$, read from the N neighbours that have a value to communicate and then, assumed the deviation $\xi_{ij} = x_{ij} - \bar{x}_i$, computes the mean square deviation:
$$s = \sqrt{\frac{\sum_{j=1}^{N} \xi_{ij}^2}{N}}.$$
2. For each neighbour j, if $|x_{ij} - \bar{x}_i| > maxError$ then $\beta_{ij} = \beta_{ij} + 1$, else $\alpha_{ij} = \alpha_{ij} + 1$.
3. If the recommendation mechanism is enabled, for each neighbour j, node i receives from any other node k in the neighbourhood the pair $(\alpha_{kj}, \beta_{kj})$, and then computes the following recommended values:
$$\alpha_j^{\text{rec}} = \sum_{1 \leq k \leq N, k \neq i,j} \frac{2 \cdot \alpha_{ik} \cdot \alpha_{kj}}{(\beta_{ik} + 2) \cdot (\alpha_{kj} + \beta_{kj} + 2) + 2 \cdot \alpha_{ik}}$$
$$\beta_j^{\text{rec}} = \sum_{1 \leq k \leq N, k \neq i,j} \frac{2 \cdot \alpha_{ik} \cdot \beta_{kj}}{(\beta_{ik} + 2) \cdot (\alpha_{kj} + \beta_{kj} + 2) + 2 \cdot \alpha_{ik}}$$
otherwise α_j^{rec} and β_j^{rec} are set to zero.
4. For each neighbour j, node i computes:
$$\alpha_j = \alpha_{ij} + \alpha_j^{\text{rec}}$$
$$\beta_j = \beta_{ij} + \beta_j^{\text{rec}}$$
5. For each neighbour j, if $\mathrm{E}(\text{Beta}(\alpha_j + 1, \beta_j + 1)) < tth$ then x_{ij} is discarded.

6. Node i computes its local value on the base of the non-discarded x_{ij}.

Notice that in order to preserve the nature of Aggregate Computing, each node computes locally and makes decisions deriving from the knowledge of its neighbourhood. The novelty is the application of a mechanism used in trust systems to monitor the neighbourhood and detect potential suspicious behaviours. The algorithm is parameterised by the two thresholds *maxError* and *tth*, which deserve empirical evaluation, as they characterise the attitude of the node to trust perturbed values and other nodes sharing perturbed values, respectively. Analogously, the recommendation mechanism represents another option that can be activated to take advantage of the knowledge shared by other (trusted) nodes. Finally, the generalisation to non-numeric field domains is possible without changing the nature of the approach and by adapting the semantics of the functions and operators used in the algorithm above.

5. Trust Implementation in ScaFi

In order to study the trust algorithm proposed in Section 4.1, we have applied it to the case of a gradient computation. The simulation framework and the experimental results are detailed in Section 6. Here, we describe the ScaFi implementation of the core trust logic in action, working as a high-level formal specification of the proposed solution, a ready-to-use JVM-based implementation, and as fully-reproducible simulation code.

The gradient algorithm presented in Section 2 can be extended to use trust as follows (new lines are highlighted):

```
def gradient(source: Boolean, trust: TrustMechanism): Double =
  rep(Double.PositiveInfinity){ distance =>
    mux(source) { 0.0 } {
      val trustMetrics = trust.metrics(distance)

      foldhoodPlus(Double.PositiveInfinity)(Math.min)(
        trust.eval(trustMetrics,
          whenTrusted = nbr { distance } + nbrRange,
          whenDistrusted = Double.PositiveInfinity
        ).value)
    }
  }
```

A trust mechanism can be thought of as a building block to be invoked at particular stages of the gradient computation, and whose interface (expressed as Scala trait) can be of the kind:

```
trait TrustMechanism {
  type Metrics
  case class EvalResults(value: Double, trusted: Boolean)

  def metrics(value: Double): Metrics
  def eval(metrics: Metrics, whenTrusted: Double, whenDistrusted: Double): EvalResults
}
```

In particular, two main phases of the gradient algorithm can be recognised:

1. *Data collection* – when the contributions of the neighbours are retrieved: at this point, the "metrics" for trust evaluation can be computed;

2. *Minimisation* – when the contributions of the neighbours are used to deduce the new gradient value by applying the triangle inequality constraint: here is when trust can be applied in order to filter out the contributions from distrusted neighbours.

Trait `BasicTrustMechanism` (Figure 3) partially implements the `TrustMechanism` contract to provide the scaffolding of a trust mechanism that uses the mean square deviation of some value to derive trust profiles and an evaluation strategy based on the beta distribution. The concrete implementations of such basic trust mechanism, with and without recommendations (class `PlainTrust` and `TrustWithRecomms`, resp.), are shown in Figure 4. The idea

```
trait BasicTrustMechanism extends TrustMechanism { self: AggregateProgram with Env with Lib =>
  case class BulkMetrics(s: Double, xmean: Double)
  case class TrustParams(a: Double, b: Double, numObservations: Int)
  case class TrustProfile(nbrId: ID, params: TrustParams)

  def bulkMetrics(value: Double): BulkMetrics = {
    def nbrVal = nbr { value }

    val n = countHood(nbrVal.isFinite)
    val sumValues = sumHood(mux(nbrVal.isFinite) { nbrVal } { 0.0 })
    val xmean = sumValues / n
    val sumSqDev = sumHood(mux(nbrVal.isFinite) { Math.pow(value - xmean, 2) } { 0.0 })
    val s = Math.sqrt(sumSqDev / n)

    BulkMetrics(s, xmean)
  }

  type MutableField[T] = MMap[ID,T]
  type AlfaBetaPair = (Double, Double)
  type AlfaBetaHistory = List[AlfaBetaPair]
  def MutableField[T](): MutableField[T] = MMap[ID,T]()

  def trustProfile(field: Double, bmetrics: BulkMetrics): TrustProfile = {
    val BulkMetrics(s: Double, xmean: Double) = bmetrics
    val (nbrId, nbrVal) = nbr{ (mid(), field) }
    val deviation = Math.abs(nbrVal - xmean)
    val maxError = Math.max(s, errorLB)

    val m = rep(MutableField[AlfaBetaHistory]()){ m => m }

    val history = m.getOrElse(nbrId, List())
    val obsEval = if(deviation > maxError) (0.0, 1.0) else (1.0, 0.0)
    m.put(nbrId, (obsEval :: history).take(observationWindow))

    val obss = m.getOrElse(nbrId, List())
    val (a,b) = obss.foldRight((0.0,0.0))((t,u) => (t._1+u._1, t._2+u._2))
    TrustProfile(nbrId, TrustParams(a, b, obss.size))
  }

  override def eval(tmetrics: Metrics, whenTrusted: Double, whenDistrusted: Double) = {
    val TrustParams(a, b, numObservations) = trustParams(tmetrics)
    val trustValue = beta(a, b)
    val isTrusted = if (numObservations >= minObservations) trustable(trustValue) else true
    val value = mux(isTrusted) { whenTrusted } { whenDistrusted }
    EvalResults(value, isTrusted)
  }

  def trustParams(m: Metrics): TrustParams // ABSTRACT

  def beta(a: Double, b: Double) = (a+1)/(a+b+2)

  def trustable(trustValue: Double): Boolean = trustValue >= trustThreshold
}
```

Figure 3: Implementation, in ScaFi, of the scaffolding for trust mechanisms adopting the beta distribution.

```
class PlainTrust extends BasicTrustMechanism { self: AggregateProgram with Env with Lib =>
  override type Metrics = TrustMetrics
  case class TrustMetrics(bmetrics: BulkMetrics, trustProfiles: Map[ID,TrustProfile])

  override def trustParams(m: TrustMetrics): TrustParams = m.trustProfiles(nbr{mid}).params

  override def metrics(value: Double): TrustMetrics = {
    val bmetrics = bulkMetrics(value)
    TrustMetrics(bmetrics, mapHood{ trustProfile(value, bmetrics) } )
  }
}
```

```
class TrustWithRecomms extends BasicTrustMechanism { self: AggregateProgram with Env with Lib =>
  override type Metrics = RecommMetrics
  case class RecommMetrics(bmetrics: BulkMetrics, nbrTrustProfiles: Map[ID,Map[ID,TrustProfile]])

  override def metrics(field: Double): RecommMetrics = {
    val bmetrics = bulkMetrics(field)

    val localTrustProfiles: Map[ID, TrustProfile] = mapHood { trustProfile(field, bmetrics) }

    RecommMetrics(bmetrics, mapHood{ nbr(localTrustProfiles) })
  }

  override def trustParams(m: RecommMetrics): TrustParams = {
    def localTrustProfiles = m.nbrTrustProfiles(mid)
    val nbrId = nbr { mid() }
    val (aRec: Double, bRec: Double) = m.nbrTrustProfiles
      .mapValues(_.get(nbrId).map(p => (p.params.a, p.params.b)).getOrElse(0.0, 0.0))
      .foldLeft((0.0, 0.0))((acc, value) => {
        // i = mid, j = nbrId ; a_j and b_j calculated from all nbrs k != i,j
        val TrustParams(a_ik, b_ik, _) =
          localTrustProfiles.get(value._1).map(_.params).getOrElse(TrustParams(0.0, 0.0, 0))
        val (a_kj, b_kj) = (value._2._1, value._2._2)
        val denom = (b_ik + 2) * (a_kj + b_kj + 2) + 2 * a_ik
        (acc._1 + 2 * a_ik * a_kj / denom, acc._2 + 2 * a_ik * b_kj / denom)
      })
    val localParams = localTrustProfiles.get(nbrId).
                                        map(_.params).getOrElse(TrustParams(0.0, 0.0, 0))
    TrustParams(localParams.a + aRec, localParams.b + bRec, localTrustProfiles.size)
  }
}
```

Figure 4: Implementation, in ScaFi, of plain and recommendations-based trust algorithms.

behind the code design is the following: a first step is to collect as much context information as possible from the neighbourhood in order to elicit a reference system (`bulkMetrics`); then, for each neighbour, a trust profile is delineated (`trustProfile`) by "reading" the individual contribution against the bulk metrics; after that, a set of trust parameters for a neighbour profile is computed (`trustParams`) and used to calculate the trust score (`beta`), i.e., the trust field; finally, the trust score is checked against a threshold to choose whether or not the currently examined neighbour has to be trusted (`trustable`), resulting in the choice of `whenTrusted` or `whenDistrusted` values for the `EvalResults` to be returned. The code makes use of some utility functions: `countHood` counts for how many neighbours the given predicate is true; `sumHood` sums the neighbours' values for the provided expression; and `mapHood` returns a map from neighbours' `ID`s to the corresponding values of the provided expression.

The key differences between `PlainTrust` and `TrustWithRecomms` lie in how the trust metrics and parameters are computed. For the latter, notice how the `localTrustProfiles` are gathered from the neighbours.

It is worth noting that, though correctly implementing the desired approach, the presented solution has a drawback: it requires the definition of a *new* gradient algorithm that is aware of, or depends on, the provided `TrustMechanism`; ideally, there should be a way to inject the logic of trust into an existing algorithm, as a sort of orthogonal component. This can be considered as an interesting future work. A potential approach would be to work at the value-tree level of aggregate programs, by defining injection points for inputs and outputs of the trust component, in a way similar to aspect-oriented programming.

6. Experimental Analysis and Evaluation

In this section, we evaluate – through simulations – the ability of the trust-based gradient implementation to respond to attacks, with and without recommendations. Specifically, we first describe the experimental framework and the simulation scenario under study (Section 6.1); then, we carry out a sensitivity analysis to study how the system behaviour is affected by changing different simulation parameters (Section 6.2); finally, we perform a comparative analysis of the effectiveness of "plain trust" and "trust with recommendations" in gradients under attack for different scenario configurations (Section 6.3).

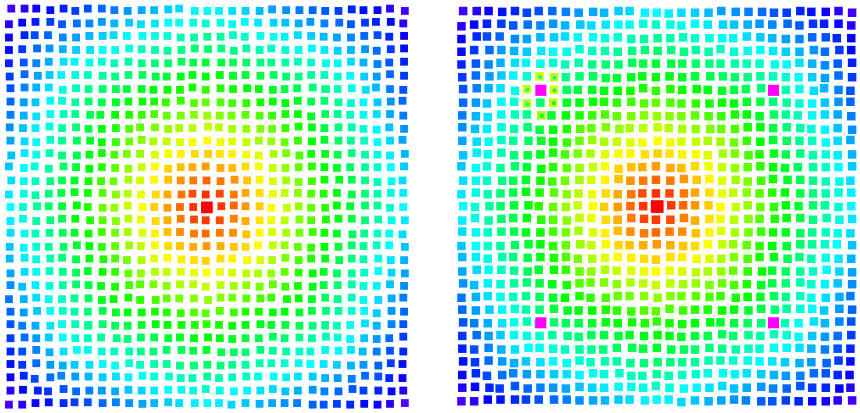
6.1. Experimental framework

We have arranged a set of experiments to exercise the trust-based gradient implementation illustrated in Section 5. These experiments take the form of simulations[4] implemented using the Alchemist simulator framework [31] and the corresponding ScaFi incarnation [20]. Alchemist supports the declarative specification of simulations as well as their execution and data collection strategy: it enables the description of topologies of nodes (according to some network model and position system), the assignment of actions to nodes (dispatched according to some time distribution), and the specification of simulation parameters (inputs) and exported values (outputs). However, to be executable, the abstract chemical-oriented meta-model provided by Alchemist has to be instantiated into a concrete model: the ScaFi incarnation does exactly that, binding Aggregate Computing abstractions into the simulation framework and giving access to the ScaFi framework for expressing and running aggregate computations.

As a first step in the modelling of the experiments, it should be noted that the effectiveness of a gradient that adopts the suggested trust algorithm can be significantly affected by many different and possibly interacting factors, such as:

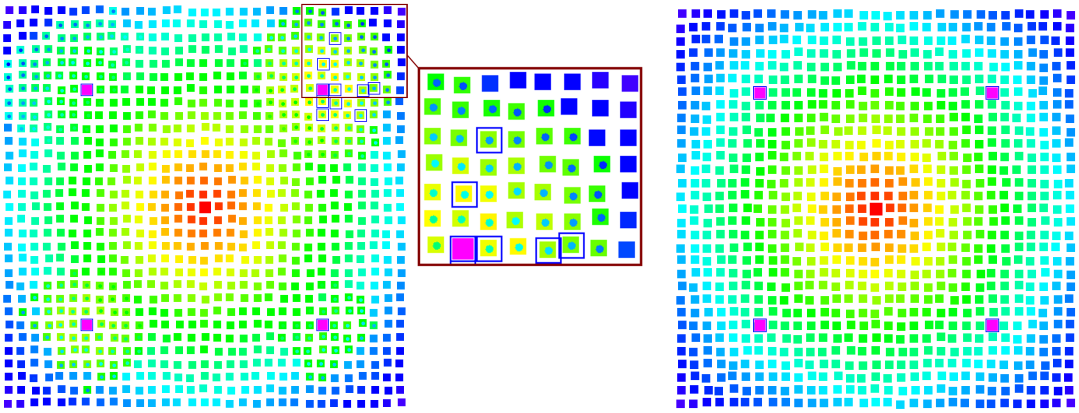
- the threshold (e.g., function of the mean square deviation s) that determines whether to deliver penalties and rewards;
- the trust threshold *tth* used to actually distrust or not a neighbour based on the locally computed trust score;
- the number of observations to be considered when computing parameters α and β, which is at the base of the ageing mechanism;
- the sophistication of the attacks;
- the topology of the network (e.g., connectivity degrees, the position of fakes); and

[4]The experimental setup is available at the following repository: `https://bitbucket.org/metaphori/trusted-ac-experiments`.



(a) Gradient field.

(b) Fake nodes appear.



(c) Trust algorithm at work while the gradient is partially corrupted.

(d) Fakes are distrusted and the gradient field is recovered.

Figure 5: Phases of the experiment. The hued squares denote the trust-based gradient field; the fuchsia squares denote fake nodes issuing random distance estimates; the little hued circles visible within the squares, e.g. in the zoomed section of (c), denote the plain gradient field which does not consider fakes (the ideal situation against which the error is calculated). Note, in (c), the corrupted gradient shape, especially at the corners (e.g., the presence, on a light square, of a darker dot means the fake is punctually creating an unwanted depression). The blue squared contours identify the nodes which are distrusted by at least one of their neighbours.

- the timing in which events take place (e.g., level of round synchronism between nodes, or the point in time where attackers appear or trust is activated).

These parameters are studied in detail for this particular case study in Sections 6.2 and 6.3. Moreover, there is a trade-off between prudence and reactivity that should be evaluated on a per-application basis.

The simulation scenario, which is meant to model a pervasive computing system, is defined as follows: there are nearly 1000 nodes (a reasonably dense computational fabric), randomly arranged in a slightly irregular grid (so as to break symmetry and check functionality with different placements and neighbourhoods); the source node is located at the centre of the grid; the fake nodes, configured to produce a random value from 0 to a maximum value set equal to the network diameter, appear at some configurable simulation time, around the source, at a configurable distance (we consider three levels from near to far). The classic gradient algorithm (i.e., the one with no trust mechanisms at work) is computed in two flavours: both without considering (G_{ideal}) and taking into account (G_{fake}) fake nodes—these should represent the optimal and worst-case gradient fields, used as the basis for calculation of the relative errors. The trust-based gradient algorithms G_{trust} and G_{recomm} are also executed, and their performance is evaluated by calculating time-wise the cumulative error given by the absolute difference between their gradient field and G_{ideal}; for G_{fake}, G_{trust},

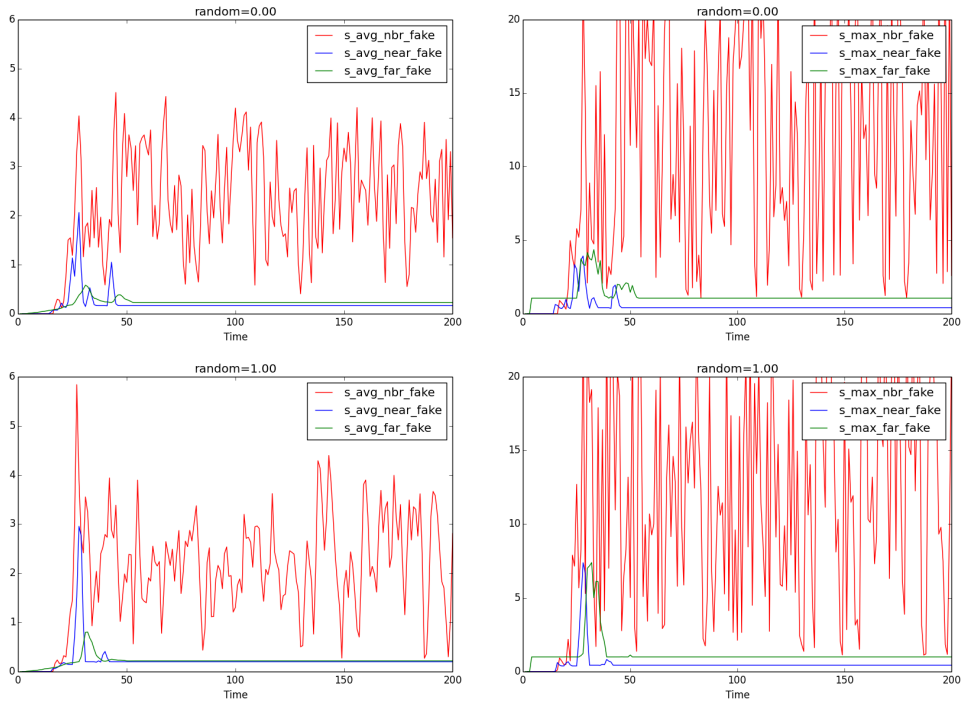


Figure 6: The fake nodes appear at $t = 20$. The graphs represent how the mean (left column) and max (right column) of s varies across time, in two different scenarios (random seeds), for three exclusive groups of nodes: the neighbours of the fake (red line), the neighbours of the fake's neighbours (blue line), and the others (green line).

and G_{recomm} such error is respectively called *err-fake*, *err-trust*, and *err-recomm*. We expect G_{trust} and G_{recomm} to behave exactly as G_{ideal} when there are no fakes (i.e., *err-trust* = *err-recomm* = 0) and, when they appear, to produce a peak of error (still lower than *err-fake*) that is progressively reduced, tending again to zero error by adjusting the field back to G_{ideal} (convergent behaviour). The simulation is run multiple times with different random seeds affecting both the particular scenario to test and the event dispatching. Figure 5 depicts the key events and phases of the simulation (in the case of fake nodes at a medium distance from the source).

6.2. Sensitivity analysis

A fundamental part of the experiment involves understanding how the different simulation parameters and design choices affect each other and the behaviour of the trust mechanism at work.

In the proposed trust algorithm, a crucial role is played by the mean square deviation s: such value is expected to provide a reference point for evaluating observations from different neighbours and thus be able to discriminate between "good" (reasonable) and "bad" (deviant) messages. Remember that, in Aggregate Computing, sharing messages is the only possible (inter)action through which the behaviour of a neighbour can be directly assessed. Figure 6 shows how s is indeed sensible to the activity of the attackers, by registering higher values in the proximity of fake nodes.

Once it has been proved that s captures the effects of punctual random attacks, the next step is to identify a function of s to be used as a *penalty threshold* for evaluating the deviation of observations and thus deciding whether to punish or reward an interaction:

```
val deviation = Math.abs(nbrVal - mean)
val maxError = ???
if(deviation > maxError) /* PUNISH */ else /* REWARD */
```

Simple kinds of functions that have been considered include:

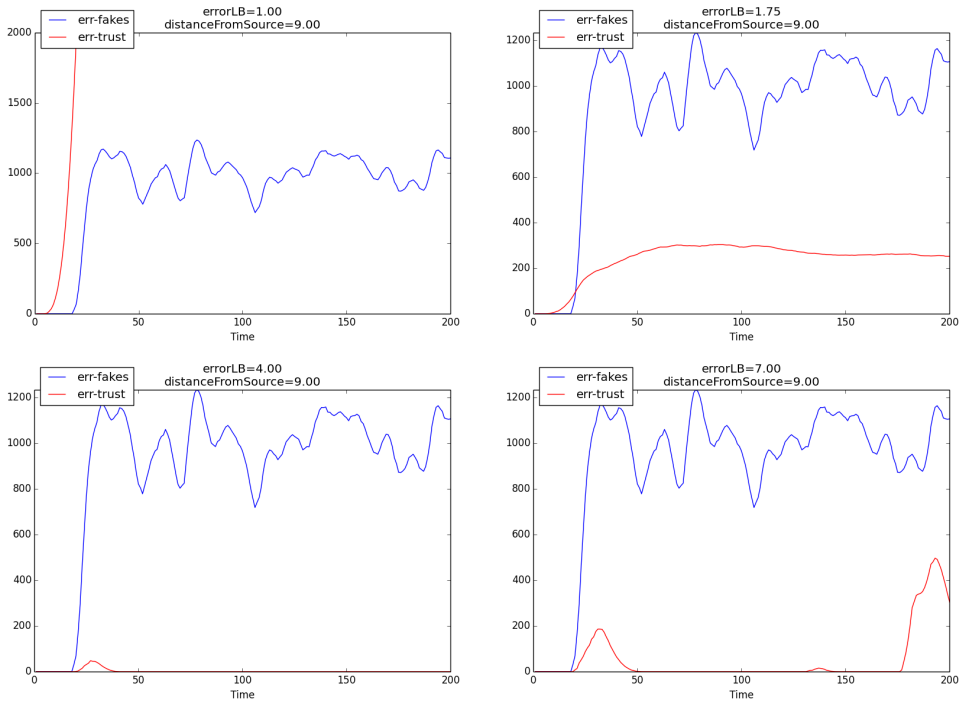


Figure 7: Calibration of *errorLB*, the lower bound for the penalty threshold. Configuration: fake at medium distance from source; *trustThreshold* = 0.75.

- Linear function of s: $maxError = k \cdot s$ — it does not work for any fixed value k, since it is unable to adequately adapt the penalty threshold to both low and high values of s.
- Step function of s: $maxError = max(s, errorLB)$ — it works by lower bounding the maximum tolerable error with both s and a constant *errorLB* parameter; in practice, it has been observed that this simple function is sufficient to avoid little deviations to count as errors and still catch disproportionate fake messages.

The calibration of *errorLB* can be driven by simulations exploring the response of the trust-based gradient, as shown in Figure 7: when *errorLB* is too low, a divergent behaviour is observed, since little tolerance is exerted, so even benevolent nodes may be penalised and hence distrusted; increasing *errorLB* can reduce the error up to a certain level or even enable the convergence of the behaviour, but once the threshold gets too high, the excessive permissiveness may fail to catch attacks and hence the error spreads in the system. Moreover, the penalty threshold needs to be aware of data deviance patterns (which are by their very nature application-specific) in order to distinguish problematic data from "normal" variability.

Also, notice that if a good calibration of *errorLB* is performed and solid guarantees exists on the stability of the scenario (e.g., fixed topology), it is possible to adopt a constant function, i.e., $maxError = errorLB$, and basically obtain the same results as with the step function of s; however, this might be fragile in general, since maintaining the same level of severity (or intensifying it, as we proved in the case of a function inversely proportional to s) in cases where the standard deviation of values is notable can result in a mass distrust of legitimate nodes.

It is also worth noticing that, ideally, the tuning of the penalty threshold should be redone after the change of any important parameter. As an extension, it is also possible to make the algorithm auto-tunable by collecting metrics (e.g., about s) and adjusting parameters (e.g., *errorLB*) accordingly; the tuning preparation phase may require to summarise information of possibly large parts of the network, so if overhead is an issue, this process could be activated right after the deployment of an application or might be triggered by a monitoring component.

6.3. On the convergence of gradients adopting Trust and Recommendations

A set of experiments has been carried out to evaluate the effect of trust and recommendations on the ability to recognise and exclude fake nodes from the computation of gradients. Specifically, the goal of this study is to assess the effectiveness of the proposed trust mechanisms as well as to understand how their functional and non-functional performance change by varying environmental or configuration parameters.

Timing of events — The simulations have been parameterised with the simulated time instants at which fake nodes enter the system or trust algorithms start working. This allows us to check for different important aspects of the dynamics. First of all, starting trust before any fake is present is useful to verify that the algorithms behave correctly in the absence of attacks (i.e., that they do not tamper with proper, unharmed evolution of the system). Secondly, by starting trust after the appearance of fakes, it is possibly to check if the system can be restored to health even after attacks have not been contrasted. In addition, it gives a practical way to check for reactivity by approximatively regulating the amount of trust input that can be collected before attacks are issued.

Trust threshold — Assuming each node is able to evaluate the individual neighbours' contributions (see Section 6.2 for details on the study of the penalty threshold) and derive from them a trust score (e.g., following a beta distribution as explained in Section 4), the decision to trust or not a given neighbour depends on another parameter, the trust threshold. Of course, the trust threshold is fundamental to the trust process as it embodies the trade-off between severity and allowance, and has to be chosen in a way to deliver as much reactivity as possible. A good value for the trust threshold should also take into account the probability of outliers and provide some margin for tolerance.

The consequences of varying the trust threshold on the gradient error are illustrated by Figure 8. First of all, notice the flash error peak, immediately absorbed, signalling the start of the trust mechanism—this could be mitigated by increasing the minimum number of observations that have to be taken before actually computing trust scores. When the fake node appears, there is a phase characterised by a variable error before convergence to the ideal field is attained; at this stage, *err-trust* and *err-recomm* may grow for some time like *err-fake*, depending on the reactivity with which the fake is discriminated. It is worth noticing how a severe trust threshold (e.g., $tth = 0.95$ or higher) penalises the plain trust but not recommendations. Indeed, recommendations provide more stability at the expense of higher initial error—this is motivated by the amplifying effect resulting from the sharing of trust values.

In general, higher values of the trust threshold (i.e., more intransigence) provide more reactivity, since fewer penalties have to be registered before distrust decisions are taken; however, once the circumspection reaches paranoid levels (i.e., *tth* is too high), too many nodes get distrusted and the system becomes unable to precisely isolate and resolve the problem. On the other hand, low values of the trust threshold correspond to a more permissive attitude; such eagerness to trust neighbours may lead to accept too many suspicious messages, resulting in a large time frame in which an attacker can operate undisturbed. In a nutshell, attitudes that are too much prudent or too much impulsive may be detrimental: the right balance of the two should be identified for any particular application. Moreover, using recommendations provides additional reactivity with respect to plain trust, since more observations are gathered in the same time frame.

Finally, employing substantial severity (or tolerance) in *both* the trust threshold and the penalty threshold is detrimental, as it becomes easy to fall into exaggerated attitudes. By contrast, varying the two thresholds in opposite directions may enable the maintenance of a balanced attitude while still taking advantage of the improved reactivity. This explains why, especially with recommendations, it is possible and indeed useful to adopt higher values of *tth*, provided that *errorLB* is large enough: prudence is used when assigning penalties, but when this happens, penalties are propagated and the sensibility of the trust threshold quickly enables distrust.

Size of the observation window — The approach adopted in this paper for calculating the trust score works by keeping track, for each neighbour, of the number of its positive and negative contributions. But how many observations have to be considered? Or, in other words, what is the size of the window of observations that provides the better performance? Here, there is a fundamental trade-off between reactivity and correctness of evaluations: if the window is too large, then it takes considerable time before changing opinion about trust; viceversa, if the window is too narrow, then decisions about mis/trusting may be too impulsive, possibly sanctioning nodes for extemporaneous situations (and this effect is accentuated by recommendations). The answers to these questions are fundamentally application-specific and depend, among other things, on the frequency of interactions, the potential impact of attacks and the speed of escalation. Figure 9 illustrates the error for different window sizes: the lack of precision of the estimates, vitiated by short windows, is amplified by recommendations and results into higher initial error, without jeopardising the convergence.

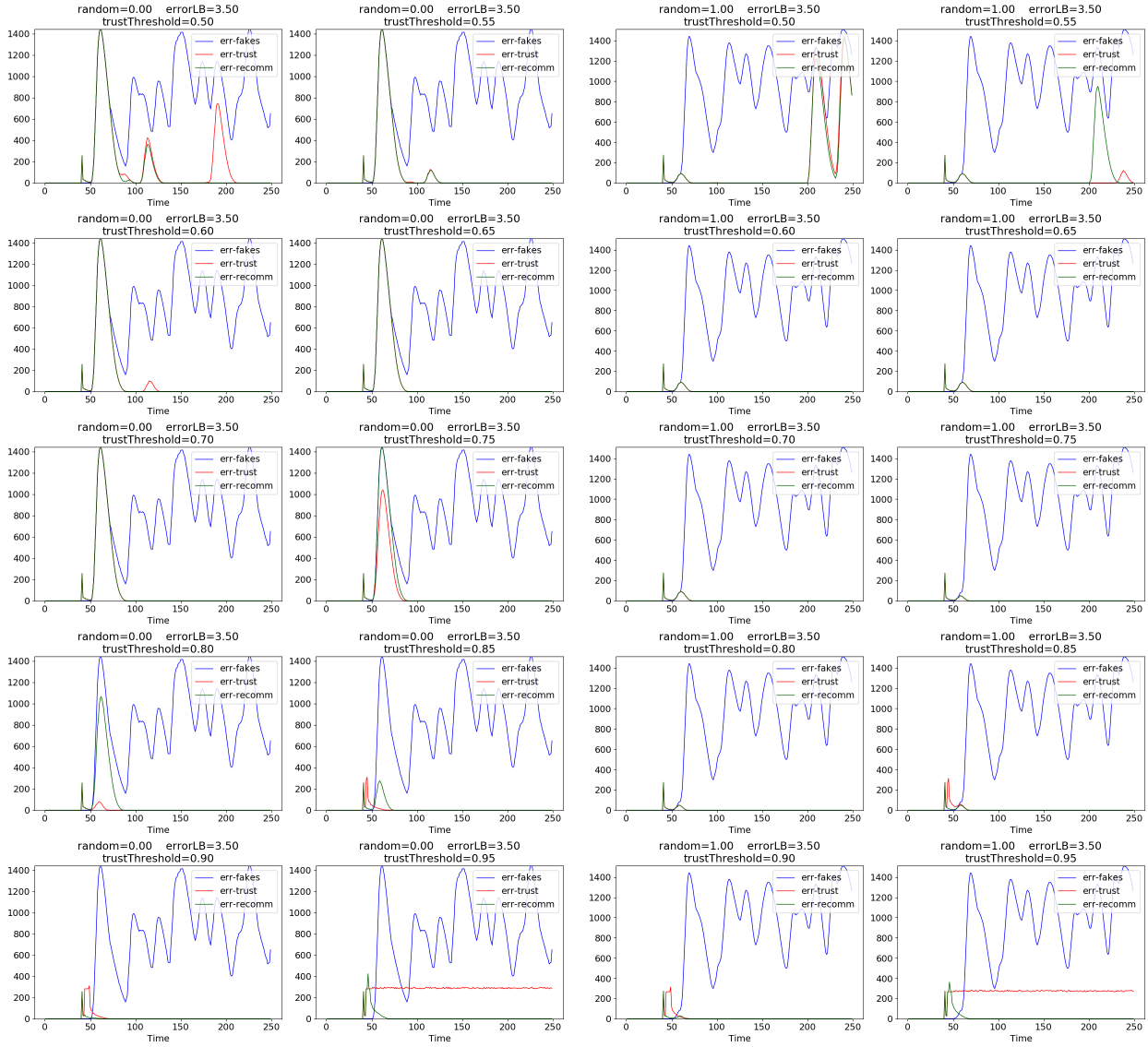


Figure 8: Experiment results for different values of *trustThreshold* and two different runs (random seeds–left and right columns, resp.). Configuration: fake at medium distance from source, appearing at $t = 50$; trust algorithm starting at $t = 40$; *errorLB* = 3.50.

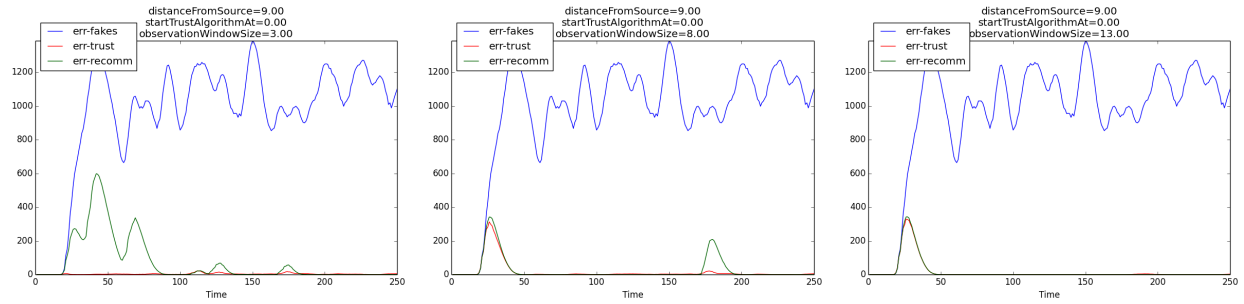


Figure 9: Errors for different observation window sizes. Configuration: fake at medium distance from source, appearing at $t = 20$; trust algorithm starting at $t = 0$; *trustThreshold* = 0.75; *errorLB* = 3.50.

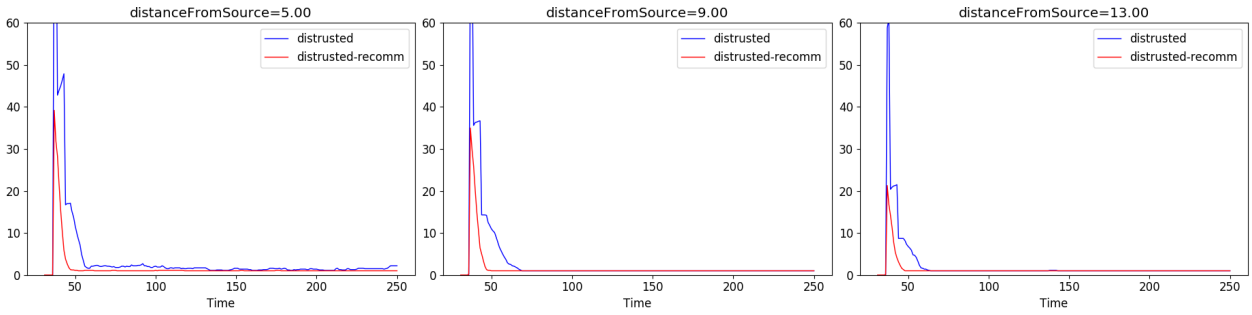


Figure 10: The graphs depict the number of nodes distrusted by at least one of their neighbours for plain trust (blue line) and recommendations (red line), for varying distances of the fake from the source. Configuration: fake appearing at $t = 20$; trust algorithm starting at $t = 30$; *trustThreshold* = 0.75; *errorLB* = 3.0.

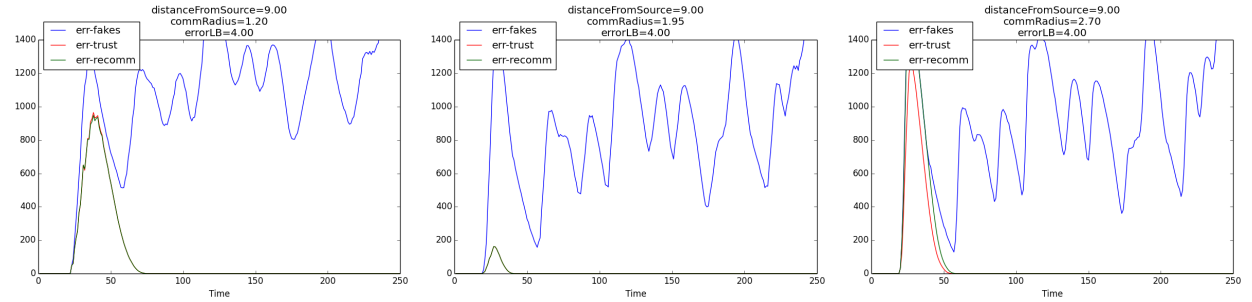


Figure 11: These graphs represent the errors committed by trust-based gradients for different values of the communication radius. Configuration: fake at medium distance from source, appearing at $t = 20$; trust algorithm starting at $t = 0$; *trustThreshold* = 0.75; *errorLB* = 4.0.

Number of distrusted nodes — By their very nature, recommendations have the effect of amplifying the consequences of trust tendencies: they potentially reduce the chance of "cases in the middle", i.e., of having nodes at the boundary between trust and distrust (where the assignment to one of the two groups would depend on contingencies). This general attitude is coupled with another key feature: the redundancy given by collecting the opinion or advice from (possibly many) other nodes. One potential effect of such behaviour is represented by Figure 10 where, with respect to plain trust, the number of nodes that get distrusted is lower.

Communication radius — The communication radius (*commRadius*) is the parameter used to regulate the extent of broadcasts from a node. Together with the position of the nodes, it concurs to define the topology of the network: the neighbours of a node δ are those nodes that lie within distance *commRadius* from δ. The effect of varying the communication radius is illustrated in Figure 11. When varying within a reasonable range (e.g., large enough to ensure basic connectivity and short enough to limit the action range of the fake nodes), there is no impact on convergence. However, it does affect the initial error: when *commRadius* is short, less neighbours are available, which means less information and higher risk for errors; similarly, when *commRadius* is large, the fakes can reach more nodes and hence create more disturbance.

Mobile fake nodes — A more in-depth study of the resilience afforded by the trust algorithms against more sophisticated attacks can be considered as a future work. However, in this paper, a first step in such direction has been taken by measuring the ability of the system to tolerate mobile attackers, i.e., fake nodes that move around the network and hence are able, with respect to the scenario with static attackers, to impact on a larger number of nodes and perturbate larger portions of the field's structure and dynamics. In this scenario, quickly identifying attackers can make a huge difference by preventing errors to cumulate and propagate. The results, for varying speeds of the fakes, are reported in Figure 12: convergence is attained only for null speed (static scenario), but the error is reduced in any case, though the ability to do so diminishes with faster speeds. Also, notice how mobility makes the dynamics more chaotic.

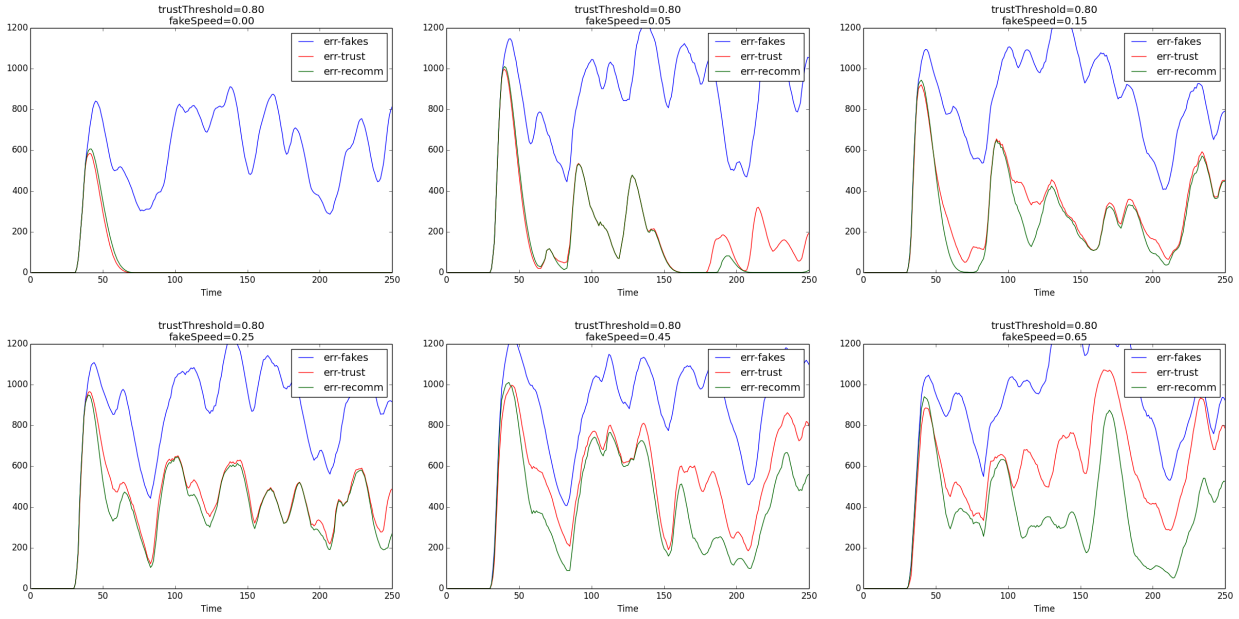


Figure 12: Error in the gradient computation, with and without using trust mechanisms, when attacked by mobile fakes producing random values. In the simulated scenario, the source is at the centre of the 2D environment, whereas the fake node moves back and forth from the upper right corner to the lower right corner and viceversa, with some slight random horizontal detour. Configuration: *trustThreshold* = 0.80; *errorLB* = 5.0.

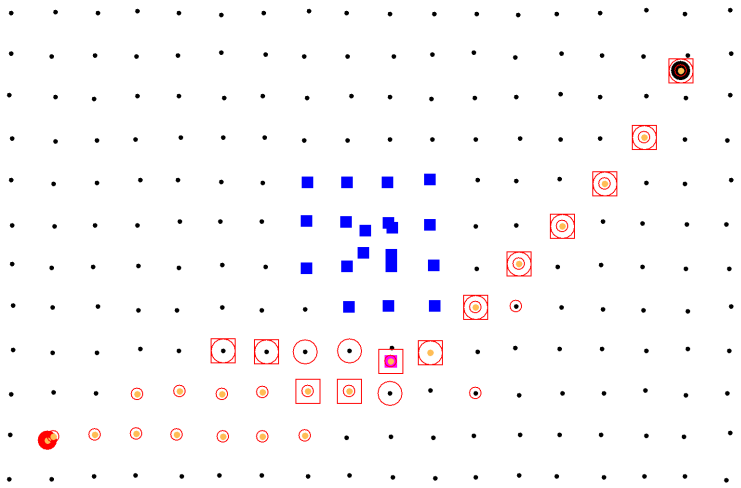
7. Case Study: Attack-Resistant Channel

The channel algorithm is a reusable building block for computing, in a distributed way, a self-healing path from a source to a destination area; it has been described in Section 2.4.2, where an implementation accepting a gradient function as input has also been presented. In this case study, the goal is to evaluate how the gradient algorithm under attack is able to support the formation of a correct channel. In practice, the following channel fields are computed:

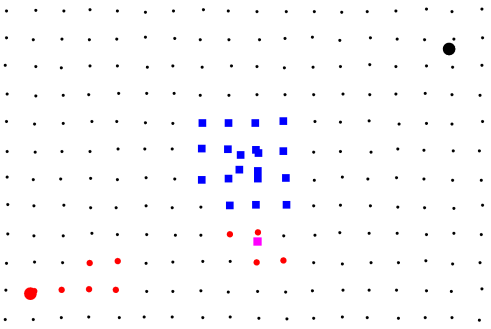
```
val cIdeal   = channel(isSrc, isTarget, width, gradient(_))
val cFake    = channel(isSrc, isTarget, width, gradient(_, fake=true))
val cTrust   = channel(isSrc, isTarget, width, gradient(_, fake=true, trust=PlainTrust))
val cRecomms = channel(isSrc, isTarget, width, gradient(_, fake=true, trust=Recommendations))
val (errFake, errTrust, errRecomms) = (cIdeal ^ cFake, cIdeal ^ cTrust, cIdeal ^ cRecomms)
```

and the error in the presence of fake contributions is measured by counting, with respect to the "right channel" in which no fakes are activated (`cIdeal`), the number of nodes with inverted boolean values (i.e., the number of `true` nodes in the XOR field), for three cases: no trust (`cFake`), plain trust (`cTrust`), and trust with recommendations (`cRecomms`). The idea is that, while the fake is able – in the absence of any trust mechanisms at work – to corrupt the gradient fields beneath the channel, effectively compromising the path to a level of complete uselessness, the adoption of trust-based gradient algorithms can provide enough resiliency to neutralise the perceived effect of the attacks. Figure 13 provides a pictorial representation of the simulation scenario, the evaluation approach and the expected outcomes.

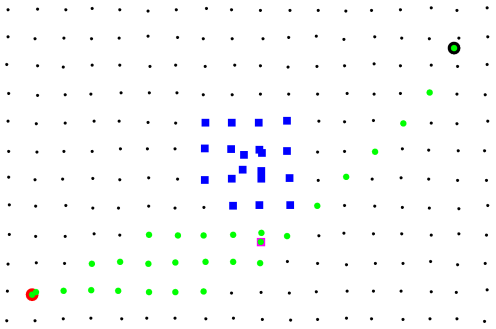
The result of the experiment for (a subset of) different runs is reported in Figure 14: when using trust, convergence is basically achieved, with the exception of some sporadic reconstructions of the channel where a noticeable error is registered during the transitory phase. Notice that, in general, the gradient and channel algorithms are self-stabilising; however, the fake node acts as a source of non-constant input and continuously perturbs them. The same experiments, launched for 2000 seconds, with *tth* = 0.94 and *errorLB* = 8.0, have shown that recommendations, after the initial error peak, maintain convergence (i.e., correct channel with null error) all the time in 17 out of 20 runs, where the



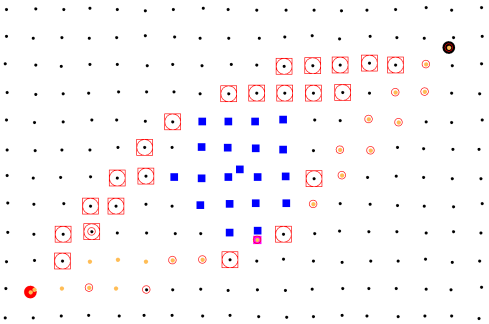
(a) Snapshot of the channel simulation at the initial stages. The big red and black nodes at the corners are the source and target nodes, respectively. Blue square nodes are obstacles. The magenta square node is the fake, which propagates random values, potentially distorting the gradient fields underlying the channel computation. Orange nodes belong to the ideal channel (i.e., the channel computed as if there were no fakes). The mere effect of the fake on the ideal channel is shown by the small red circles, which denote those nodes yielding a wrong value of channel membership; instead, the big red circles (resp., squares) are used to highlight errors committed while also *using* trust (resp., recommendations).



(b) The channel created by the fake never stabilises. This snapshot shows the channel path without using trust is completely compromised.



(c) Using trust (possibly with recommendations), it is possible to preserve the channel. Notice that it might be slightly different from the ideal channel visible in (a).



(d) Sometimes, the reduction of error provided by trust in the gradients beneath the channel algorithm may result in a different path being chosen. Though reasonable, it still counts as an error in our evaluation.

Figure 13: Snapshots from the channel simulation.

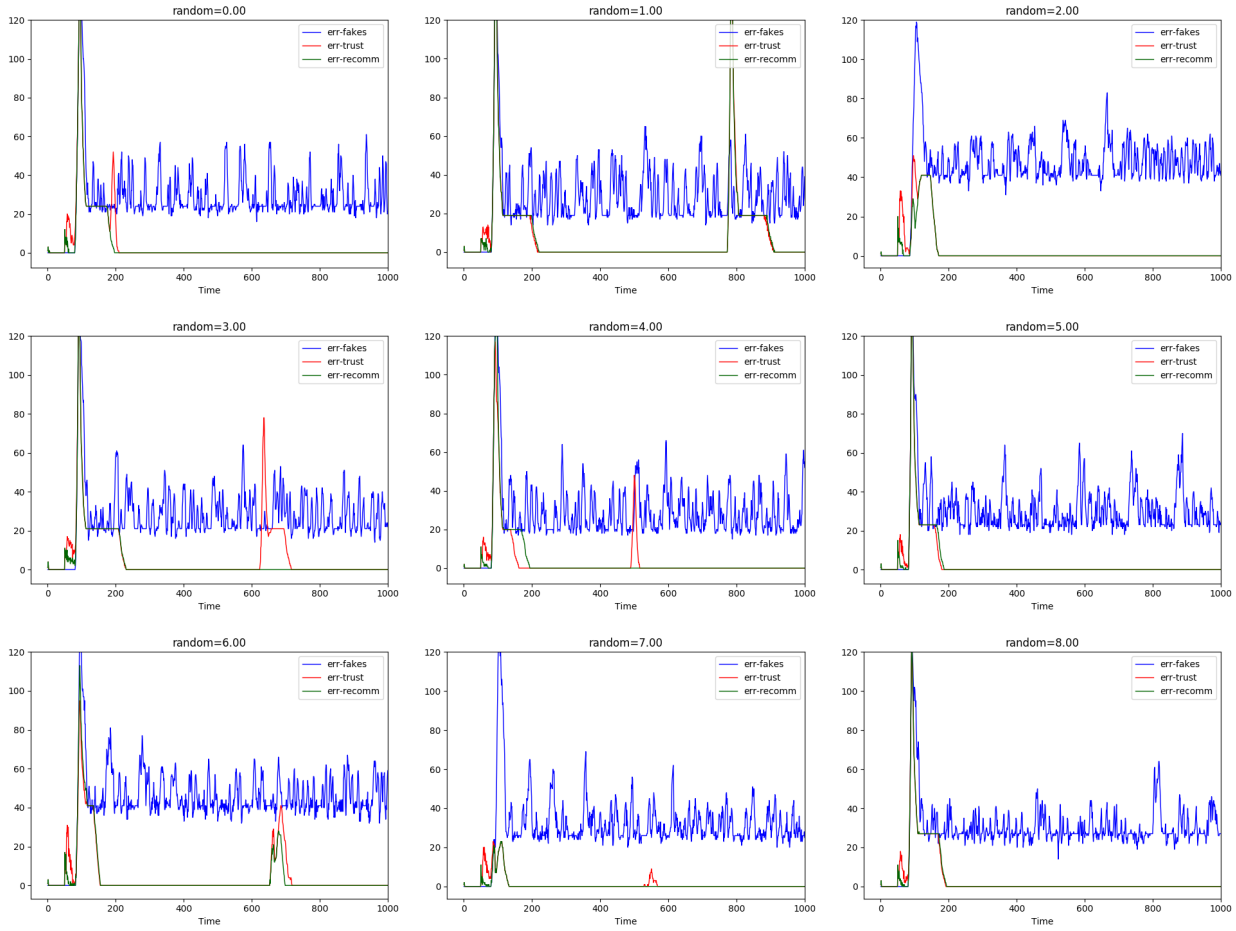


Figure 14: Each graph shows the evolution of the error across time for different random seeds. Configuration: fake appearing at $t = 80$; trust algorithm starting at $t = 50$; *trustThreshold* = 0.90; *errorLB* = 8.0.

remaining 3 runs only present one or two peaks of error that are quickly fixed, still exhibiting the correct channel for the majority of time.

In summary, the contribution is clear: gradient implementations adopting trust mechanisms improve stability and provide resistance to attacks in such a way that they do not impact on higher level building blocks (and in turn to applications).

It is important to notice that attacking the gradients underlying the channel is the way by which an attacker can impact the system the most: in fact, by trying to provoke a disruptive global effect out of merely local contributions, it is possible to completely compromise the channel. In addition, some mechanism should be used to prevent a malevolent node from pretending to be source or target. Instead, attacks directed at top-level channel values, while skipping the defence line provided by the trust-based gradient implementation, must hijack several nodes to produce a significant system-wide effect (e.g., a channel partition). Functionality might still be locally compromised, though. This aspect, which is left as an interesting future work, might possibly be tackled by enforcing invariants between different parts of an aggregate computation.

8. Related Work and Conclusion

Trends like pervasive computing and IoT foster a vision of large-scale open systems in which application services are supported through self-organisation and cooperative computations, possibly according to opportunistic dynamics.

Aggregate Computing is a promising paradigm for engineering collective adaptive systems, thanks to its abstraction, global stance, and compositionality. Approaches for programming collectives of devices have been investigated in many fields and applications, including spatial computing, multi-agent systems, wireless sensor networks (WSNs), swarm robotics, P2P systems, and so on. In fact, Aggregate Computing itself arose as a generalisation of a number of previous methods and languages, extensively surveyed in [32] where four main families are identified:

i. *device abstraction languages* (e.g., TOTA [33], Hood [34]) that simplify the programming of individual networked devices, for instance by supporting local communications;
ii. *pattern languages* (e.g., GPL [35], OSL [36]) that provide means for arranging elements according to geometric and topological patterns;
iii. *information movement languages* (e.g., TinyDB [37], Regiment [38]) for streaming and summarising data situated in space; and
iv. *general-purpose spatial languages* (e.g., MGS [39] and Proto [40]) that include elements of the other families to provide specialised support for spatial computing.

Other works share goals or traits of individual portions of the Aggregate Computing stack, particular aspects of the execution or interaction model, or implementation-level techniques: examples include BSP-inspired graph processing tools (e.g., Pregel [41]), languages for mobile ad-hoc networks (MANETs) and WSNs (e.g., SpatialViews [42]), and programming models for scalable computations (e.g., MapReduce [43]). Compared to previous approaches, and most importantly for the goal of this paper, Aggregate Computing based on the field-calculus supports a compositional model that is key to isolate the effect of attacks on given distributed components, and to study their downstream effect on larger systems, as we started studying with the *channel* example.

Security is a prominent concern in any computer-based system. Distribution and openness take along their own set of threats and vulnerabilities, and IoT scenarios are characterised by a very large attack surface. Despite the Aggregate Computing framework is resilient against many temporary sources of failure, and self-organisation can deliver some form of robustness, the security aspect is even more critical in aggregate systems. On a business perspective, the suitability of the approach to security-critical (e.g., IoT systems) and safety-critical applications (e.g., crowd management and tactical networks) makes security support paramount. By a technical point of view, the ability to drive global behaviours from local activity and the fundamental assumption of cooperation on the nodes involved – which is convenient by an algorithm design perspective – create significant risks and arise a problem of trust in aggregate computations.

Over the last decades, a lot of research was done to promote the effective use of computational definitions of notions of trustworthiness with the aim of simplifying and automatising trust-based decision making processes. Such an effort ranges from formal specification approaches [44] and verification methods [45, 46, 47, 48, 49] to the development of trust management systems and the analysis of related threats and vulnerabilities [25, 22]. In particular, decentralised trust and reputation systems are proposed as soft securing cooperation mechanisms in various kinds of distributed systems, including participatory sensing systems, crowdsourcing and sensor networks [28, 24, 50, 23, 25, 26, 51, 27, 52], P2P and user-centric systems [53, 54, 55]. The peculiarities of these systems concern the management of trust in terms of acquisition of basic information, local storage, trust estimation algorithm, and subsequent dissemination, as well as the combination of local trust and recommendations. As a novelty of the present work, such mechanisms have been applied to Aggregate Computing in order to investigate whether the adaptive and self-organising nature of local devices programmed by a global stance is compatible with decentralised systems for the estimation of trustworthiness relations.

In this paper, we have consolidated and extended the work in [8] that had for the first time considered security-related aspects in the context of Aggregate Computing. In particular, our study has focused on attacks based on the diffusion of well-formed, factitious messages and we have proposed the use of trust mechanisms to make algorithms resistant to them. After the sensitivity analysis of the proposed trust metrics, several experiments have been executed to verify the ability of an extended gradient algorithm adopting trust mechanisms to tolerate attacks issued by one or more fake nodes emitting random values capable of distorting the ideal gradient field. The investigation has also been complemented with an analysis of the effect, in terms of error and convergence, of some key influencing factors and parameters, for both plain trust and trust with recommendations. The performance analysis has shown the efficacy of the trust system and, when integrating recommendations weighted in ratio of the reputation of the recommenders, a general improvement in terms of stability and efficiency (the evolution of the error is characterised by a rapid

convergence to zero). These results are fully compatible with the analysis proposed in [24, 56], where the same recommendation system is evaluated in the distributed setting of sensor networks. In particular, analogously to our results, in [56] it is shown that second-hand information improves both efficacy and efficiency with respect to the plain trust mechanism, provided that recommendations are not blindly combined with the trust metric computed locally without any appropriate weighting mechanism. Finally, as already mentioned, through the channel case study it has been shown that using trust-based gradient algorithms can provide protection up to higher-level aggregate building blocks. In a nutshell, trust can support attack-resistant cooperative computations, which are expected to play a crucial role in, for instance, IoT scenarios.

This work has prepared the ground for interesting future works. Concerning the specific trust mechanism we proposed, it would be interesting to test the applicability of more sophisticated computational notions of trust for distributed systems to the Aggregate Computing framework, e.g., with respect to ageing and recommendation aspects. On a broader scope, an extensive investigation of the vulnerabilities of the full Aggregate Computing stack would be important to start filling the gaps for real-world adoption of these techniques. From a programming perspective, it is impractical to require the redefinition of aggregate building blocks to use trust; hence, some means to inject trust mechanisms into existing algorithms would be valuable. Last but not least, it would be interesting to understand what it takes to apply the trust approach to other scenarios and algorithms, and what results can be achieved: in this paper, the case studies have focused on self-healing gradient fields and very simple attacks; the next steps may possibly consist of a generalisation effort and a study of the ability of the approach to scale with the sophistication of the attacks.

References

[1] A. Ferscha, Collective adaptive systems, in: Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers, UbiComp/ISWC'15 Adjunct, ACM, New York, NY, USA, 2015, pp. 893–895. doi:10.1145/2800835.2809508.

[2] M. Viroli, A. Omicini, A. Ricci, Engineering MAS environment with artifacts, in: D. Weyns, H. V. D. Parunak, F. Michel (Eds.), 2nd International Workshop "Environments for Multi-Agent Systems" (E4MAS 2005), AAMAS 2005, Utrecht, The Netherlands, 2005.

[3] J. Beal, D. Pianini, M. Viroli, Aggregate programming for the Internet of Things, IEEE Computer 48 (9).

[4] J. L. Fernandez-Marquez, G. D. M. Serugendo, S. Montagna, M. Viroli, J. L. Arcos, Description and composition of bio-inspired design patterns: a complete overview, Natural Computing 12 (1) (2013) 43–67. doi:10.1007/s11047-012-9324-y.

[5] F. Damiani, M. Viroli, J. Beal, A type-sound calculus of computational fields, Science of Computer Programming 117 (2016) 17 – 44. doi:http://dx.doi.org/10.1016/j.scico.2015.11.005.

[6] F. Damiani, M. Viroli, D. Pianini, J. Beal, Code mobility meets self-organisation: A higher-order calculus of computational fields, in: S. Graf, M. Viswanathan (Eds.), Formal Techniques for Distributed Objects, Components, and Systems, Vol. 9039 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 113–128.

[7] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, D. Pianini, From field-based coordination to aggregate computing, in: G. Di Marzo Serugendo, M. Loreti (Eds.), Coordination Models and Languages, Springer International Publishing, Cham, 2018, pp. 252–279.

[8] R. Casadei, A. Aldini, M. Viroli, Combining trust and aggregate computing, in: Springer LNCS, 15th Int. Workshop on Foundations of Coordination Languages and Self-Adaptative Systems (FOCLASA17), Trento, 2017, in press.

[9] M. Viroli, R. Casadei, D. Pianini, On execution platforms for large-scale aggregate computing, in: Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct, ACM, 2016, pp. 1321–1326.

[10] V. Massimo, F. Maria, D. Schahram, R. Omer, R. Ranjan, Osmotic computing: A new paradigm for edge/cloud integration, IEEE Cloud Computing 3 (6) (2016) 76–83. doi:doi.ieeecomputersociety.org/10.1109/MCC.2016.124.

[11] M. Viroli, F. Damiani, J. Beal, A calculus of computational fields, in: C. Canal, M. Villari (Eds.), Advances in Service-Oriented and Cloud Computing, Vol. 393 of Communications in Computer and Information Science, Springer Berlin Heidelberg, 2013, pp. 114–128.
URL `http://dx.doi.org/10.1007/978-3-642-45364-9_11`

[12] M. Viroli, J. Beal, F. Damiani, D. Pianini, Efficient engineering of complex self-organising systems by self-stabilising fields, in: Self-Adaptive and Self-Organizing Systems (SASO), IEEE 9th International Conference on, IEEE, 2015, pp. 81–90. doi:10.1109/SASO.2015.16.

[13] M. Viroli, G. Audrito, J. Beal, F. Damiani, D. Pianini, Engineering resilient collective adaptive systems by self-stabilisation, ACM Transaction on Modelling and Computer Simulation 28 (2) (2018) 16:1–16:28. doi:10.1145/3177774.
URL `http://doi.acm.org/10.1145/3177774`

[14] R. Casadei, M. Viroli, Towards aggregate programming in Scala, in: 1st Workshop on Programming Models and Languages for Distributed Computing, ACM, 2016, p. 5.

[15] `https://akka.io` — Accessed: 2018-05-20.

[16] R. Casadei, M. Viroli, Programming actor-based collective adaptive systems, in: Programming with Actors - State-of-the-Art and Research Perspectives, Vol. 10789 of Lecture Notes in Computer Science, Springer, 2018, to appear.

[17] G. Audrito, F. Damiani, M. Viroli, R. Casadei, Run-time management of computation domains in field calculus, in: Foundations and Applications of Self* Systems, IEEE International Workshops on, IEEE, 2016, pp. 192–197.

[18] J. Bachrach, J. Beal, J. McLurkin, Composable continuous-space programs for robotic swarms, Neural Computing and Applications 19 (6) (2010) 825–847. doi:10.1007/s00521-010-0382-8.
URL https://doi.org/10.1007/s00521-010-0382-8
[19] G. Audrito, R. Casadei, F. Damiani, M. Viroli, Compositional blocks for optimal self-healing gradients, in: Self-Adaptive and Self-Organising Systems (SASO), IEEE International Conference on, IEEE, 2017.
[20] R. Casadei, D. Pianini, M. Viroli, Simulating large-scale aggregate MASs with Alchemist and Scala, in: Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on, IEEE, 2016, pp. 1495–1504.
[21] A. Perrig, R. Szewczyk, J. Tygar, V. Wen, D. Culler, SPINS: Security protocols for sensor networks, Wireless Networks 8 (5) (2002) 521–534.
[22] F. G. Marmol, G. M. Perez, Security threats scenarios in trust and reputation models for distributed systems, Computers and Security 28 (7) (2009) 545–556.
[23] J.-H. Cho, A. Swami, I.-R. Chen, A survey on trust management for mobile ad hoc networks, Communications Surveys & Tutorials 13 (4) (2011) 562–583.
[24] S. Ganeriwal, L. K. Balzano, M. B. Srivastava, Reputation-based framework for high integrity sensor networks, ACM Trans. Sen. Netw. 4 (3) (2008) 1–37.
[25] Y. Yu, K. Li, W. Zhoub, P. Lib, Trust mechanisms in wireless sensor networks: Attack analysis and countermeasures, Journal of Network and Computer Applications 35 (3) (2012) 867–880.
[26] G. Han, J. Jiang, L. Shu, J. Niu, H.-C. Chao, Management and applications of trust in wireless sensor networks: A survey, Journal of Computer and System Sciences 80 (3) (2014) 602–617, special Issue on Wireless Network Intrusion.
[27] H. Mousa, S. B. Mokhtar, O. Hasan, O. Younes, M. Hadhoud, L. Brunie, Trust management and reputation systems in mobile participatory sensing applications: A survey, Computer Networks 90 (2015) 49–73.
[28] S. Buchegger, J.-Y. L. Boudec, A robust reputation system for peer-to-peer and mobile ad-hoc networks, in: 2nd Workshop on the Economics of Peer-to-Peer Systems, P2PEcon, 2004.
[29] A. Jøsang, R. Ismail, The beta reputation system, in: 15th Bled Conf. on Electronic Commerce, 2002.
[30] B. Priayoheswari, K. Kulothungan, A. Kannan, Beta reputation and direct trust model for secure communication in wireless sensor networks, in: Int. Conf. on Informatics and Analytics, ICIA-16, ACM, 2016, pp. 1–5.
[31] D. Pianini, S. Montagna, M. Viroli, Chemical-oriented simulation of computational systems with Alchemist, Journal of Simulation-doi:10.1057/jos.2012.27.
[32] J. Beal, S. Dulman, K. Usbeck, M. Viroli, N. Correll, Organizing the aggregate: Languages for spatial computing, arXiv preprint arXiv:1202.5509.
[33] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications: The TOTA approach, ACM Trans. on Software Engineering Methodologies 18 (4) (2009) 1–56. doi:http://doi.acm.org/10.1145/1538942.1538945.
[34] K. Whitehouse, C. Sharp, E. Brewer, D. Culler, Hood: a neighborhood abstraction for sensor networks, in: Proceedings of the 2nd international conference on Mobile systems, applications, and services, ACM Press, 2004.
[35] D. Coore, Botanical computing: A developmental approach to generating interconnect topologies on an amorphous computer, Ph.D. thesis, MIT (1999).
[36] R. Nagpal, Programmable self-assembly: Constructing global shape using biologically-inspired local interactions and origami mathematics, Ph.D. thesis, MIT (2001).
[37] S. R. Madden, R. Szewczyk, M. J. Franklin, D. Culler, Supporting aggregate queries over ad-hoc wireless sensor networks, in: Workshop on Mobile Computing and Systems Applications, 2002.
[38] R. Newton, M. Welsh, Region streams: Functional macroprogramming for sensor networks, in: First International Workshop on Data Management for Sensor Networks (DMSN), 2004, pp. 78–87.
[39] J.-L. Giavitto, C. Godin, O. Michel, P. Prusinkiewicz, Computational models for integrative and developmental biology, Tech. Rep. 72-2002, Univerite d'Evry, LaMI (2002).
[40] J. Beal, J. Bachrach, Infrastructure for engineered emergence in sensor/actuator networks, IEEE Intelligent Systems 21 (2006) 10–19.
[41] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010, pp. 135–146.
[42] Y. Ni, U. Kremer, A. Stere, L. Iftode, Programming ad-hoc networks of mobile and resource-constrained devices, ACM SIGPLAN Notices 40 (6) (2005) 249–260.
[43] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.
[44] A. Jøsang, A logic for uncertain probabilities, Int. Journal of Uncertainty, Fuzziness, and Knowledge-Based Systems 9 (3) (2001) 279–311.
[45] A. Aldini, Design and verification of trusted collective adaptive systems, ACM Transactions on Modeling and Computer Simulation 28 (2) (2018) Article 9.
[46] A. Aldini, Modeling and verification of trust and reputation systems, Journal of Security and Communication Networks 8 (16) (2015) 2933–2946.
[47] J. Huang, A formal-semantics-based calculus of trust, Internet Computing 14 (5) (2010) 38–46.
[48] Z. Li, H. Shen, Game-theoretic analysis of cooperation incentives strategies in mobile ad hoc networks, Transactions on Mobile Computing 11 (8) (2012) 1287–1303.
[49] D. Trcek, A formal apparatus for modeling trust in computing environments, Mathematical and Computer Modelling 49 (1–2) (2009) 226–233.
[50] J. Li, R. Li, J. Kato, Future trust management framework for mobile ad hoc networks, IEEE Communications Magazine 46 (4) (2008) 108–114.
[51] A. Tarable, A. Nordio, E. Leonardi, M. G. A. Marsan, The importance of being earnest in crowdsourcing systems, in: IEEE Conference on Computer Communications, INFOCOM, IEEE, 2015, pp. 2821–2829.
[52] H. S. Packer, L. Dragan, L. Moreau, An auditable reputation service for collective adaptive systems, in: Social Collective Intelligence: Combining the Powers of Humans and Machines to Build a Smarter Society, Springer, 2014, pp. 159–184.

[53] E. Koutrouli, A. Tsalgatidou, Reputation-based trust systems for P2P applications: Design issues and comparison framework, in: Trust and Privacy in Digital Business: Third International Conference, TrustBus 2006, Springer, 2006, pp. 152–161.
[54] Y. Zhang, L. Lin, J. Huai, Balancing trust and incentive in peer-to-peer collaborative system, Journal of Network Security 5 (2007) 73–81.
[55] R. Yaich, O. Boissier, P. Jaillon, G. Picard, An adaptive and socially-compliant trust management system for virtual communities, in: 27th Annual ACM Symposium on Applied Computing, SAC, ACM, 2012, pp. 2022–2028.
[56] S. Ganeriwal, M. B. Srivastava, Reputation-based framework for high integrity sensor networks, in: Procs. of the 2nd ACM Workshop on Security of Ad Hoc and Sensor Networks, ACM, 2004, pp. 66–77.