Combining trust and aggregate computing

Roberto Casadei[1], Alessandro Aldini[2], and Mirko Viroli[1]

[1] Alma Mater Studiorum—Università di Bologna, Italy
{roby.casadei,mirko.viroli}@unibo.it
[2] Università di Urbino Carlo Bo, Italy
alessandro.aldini@uniurb.it

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 of 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 a 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 overall computation outcome and hence the algorithms built on top of it. 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. To better understand the fragility of aggregate systems in face of attacks and investigate possible countermeasures, in this paper we consider the paradigmatic case of *gradient* algorithm, analysing the impact of offences and the mitigation afforded by the adoption of trust mechanisms.

Keywords: aggregate computing; collaborative p2p systems; security; safety; trust.

1 Introduction

The last decades have been feeding a process where large numbers of interconnected computing devices get deployed in our environments. Such technological and social movements seem to imply a future of increasing pervasiveness and interconnection, where dense networks of computer-like nodes are overlaid on and tightly interacting with our physical world and humans in it. Exploiting 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 unfeasible and pushes forward the need to endow such systems with self-* properties, but hence a whole set of new challenges arises.

The field of collective adaptive systems is devoted to the study of systems where large groups of entities jointly seek to reach their goals in a dynamic environment. The main issues include (i) how to provide an effective specification of the self-organising and goal-oriented behaviour of the system and (ii) how to turn that program into resilient and efficient execution. Moreover, it should be noted that these scenarios are not only interesting by a scientific point of view, but also from the engineering side—where trade-offs have to be taken and limited resources need to be coped with. Additionally, given this complex setting and the impracticality of in-field tests, it would be crucial to have ways to obtain certain guarantees on the correctness of implementations; for this purpose, both formal methods and simulations are invaluable.

Aggregate computing [2], which we recall in Section 2, is one promising approach for the engineering of (possibly large-scale) distributed systems that need to resiliently adapt to local, environmental conditions. It takes an abstract, global stance in which one programs the desired collective behaviour, through a composition of self-organising and coordination patterns, 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 [7], which builds on the idea of computational fields, 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 collective and adaptive systems, trust represents a fundamental aspect in a setting in which the rapid and continuous exchange and propagation of information is a key feature. The need for cooperation is typically accompanied by the growth of potential (insider) security threats, which may depend on selfish or malicious behaviors 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. Over the last decades, a lot of research was done to promote the effective use of computational notions of trust allowing to estimate such a belief perception and simplify related decision making processes. Such an effort ranges from formal specification approaches [11] and verification methods [1, 10, 14, 19] to the development of distributed trust management systems [6, 16, 8, 13] and the analysis of threats and vulnerabilities [23, 15].

Starting with these considerations, this paper develops on the combination of trust and aggregate computing, providing the following contributions:

– an overview of the security issues in aggregate computing (see Section 3);
– the applicability of classical trust management techniques in this framework (see Section 4);
– an empirical study of the effectiveness of trust fields in mitigating or avoiding at all the consequences of (deliberate or not) misbehaviours of nodes in a gradient computation (see Section 5).

2 Aggregate Computing

Aggregate computing [2] is an approach for the engineering of systems where coordinated adaptation plays a major role. It is built around three interrelated core ideas.

The first idea is *working at the macro-level while letting the platform deal with the micro-macro bridging*: the target of programming is not an individual element of the system but rather the whole system itself, which can be seen as a distributed computational body, i.e., a programmable aggregate. This partial inversion of the point of view is what relieves the programmer of solving the local-to-global mapping problem by herself, i.e., what supports the steering of a self-organising process by declaring the desired outcome from the favoured intermediate position between the micro and the macro levels.

Secondly, the approach is *compositional*, in that it enables complex behaviours to be built out of simpler ones. There is a number of general coordination operators, building blocks and patterns that can be used in concert to specify a sort of global-state flow graph. The point is having one adaptive behaviour (of predictable dynamics) feed another one and so on until the proper chain of system transformations is in place.

The third idea relates to *abstraction*, namely, leaving some pieces of information unspecified, for achieving declarativeness 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 [21]—ranging from completely peer-to-peer to centralised (server- or cloud-based), up to hybrid and adaptive ones (e.g., according to available infrastructure).

2.1 Fields and aggregate computations

The key concept that allows us to put this conception into practice is that of *computational field*. This is an abstraction that maps every device to a computational value over time. If, as often is 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; this gives the primary interpretation, the digital-physical correspondence that makes this framework amenable

to programming situated collective adaptive systems. The time-wise nature of fields is what accounts for dynamics, whereas the spatial dimension provides a foundation for context and local interaction. Also note that a field can be interpreted both punctually and globally, opening the way for bridging individual and aggregate behaviour.

The formal framework for working with fields is known as the *field calculus* [7, 22]. It provides the basic constructs for transforming and combining field computations together. Hence, an aggregate program can be represented as a field expression, deployed to a set of networked devices geared with middleware aggregate support, and executed according to the operational semantics of the field calculus. A system enacting field computations has also to operate 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. By 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, which is known as "alignment," 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).

2.2 ScaFi and the field calculus constructs

ScaFi [5] (Scala Fields) is an open-source framework[3] for aggregate computing. It provides a Scala-internal Domain-Specific Language (DSL) for expressing and running aggregate computations according to the field calculus; it gives access to aggregate programming features[4] together with the type system and all the

[3] `https://bitbucket.org/scafiteam/scafi`

[4] The semantics of the field calculus has been implemented in a slightly different but largely equivalent way with respect to the "standard" one, due to design choices as well as technicalities involved in the DSL embedding.

amenities that can be found in a mainstream programming language supporting imperative, object-oriented and functional paradigms.

The core constructs for working with fields are reified as plain, generic methods[5]; they are defined in the following Scala interface[6].

```
trait Constructs {
  // Key constructs
  def rep[A](init: => A)(fun: A => A): A
  def aggregate[A](f: => A): A
  def nbr[A](expr: => A): A
  def foldhood[A](init: => A)(acc: (A, A) => A)(expr: => A): A

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

When discussing the 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, expression `5` can be thought of as either a single number calculated by a specific device or a field of fives.

Construct `rep(init)(fun)` yields an `init` field that evolves over time according to the given state transformation `fun`ction. 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 field counting the number of rounds performed (point-wise).

Construct `aggregate(f)` is used to wrap the body expression `f` of a function that has to be interpreted according to the aggregate semantics, which means that it must work as a unit for alignment; in other words, an aggregate function expresses a subcomputation which restricts the domain of the field to only the devices that are executing that very subcomputation. For example, consider how this foundational feature can be used to implement a building block `branch` that splits the domain of computation according to a boolean field `cond` (here,

[5] Actually, fields do not explicitly appear in the method signatures: there are no "first-class" fields in ScaFi; rather, that notion (which still can be used while reasoning about code) has been replaced with that of *neighbour-dependent expression*.

[6] In Scala, methods 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); syntax `(T1,T2)` denotes a 2-element tuple type; syntax `A=>R` denotes a function type; and syntax `=> R` denotes a call-by-name parameter which is passed unevaluated to the method body and in there gets (re-)evaluated at each use (basically it is a syntactic shorthand over nullary functions).

`mux` is a purely functional multiplexer; moreover, notice how Scala functions can be directly used to wrap aggregate code and hence enable code reuse in a conventional way):

```
def branch[A](cond: => Boolean)(th: => A)(el: => A): A =
  mux(cond)(() => aggregate{ th })(() => aggregate{ el })()
```

The code works be first creating a field of (anonymous) functions out of the `cond`ition field, and then using the function application operator `()` to continue the computation at the two branches. The nodes that follow the then-branch (or the else-branch, likewise) will be able to interact only with the nodes that followed the same path; i.e., the field gets split into two completely separated parts.

In the field calculus, communication is achieved through neighbouring fields. A neighbouring field can be thought of as a field of fields; point-wise, every device is mapped to a field that consists of the information shared by its neighbours. Now, construct `foldhood(init)(acc)(expr)` is what allows you to `acc`umulate the neighbouring field `expr` (using `init` as the identity field for operation `acc`) into a flat field (or to a single value in the device-centric interpretation). `nbr(expr)` is the basic operator that supports communication of information among devices[7], i.e., the creation of neighbouring fields. For example,

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

denotes the field of all the devices that sensed an alarm or that are nearby someone who 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 are a special kind of sensors, known as neighbouring sensors and read through `nbrvar`, that, similarly to `nbr`, produce neighbouring fields; they are of use for mapping each neighbour set to platform-level readings. A common one is `nbrRange`, that yields the (second-order) field of distances from neighbours:

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

Just like `nbr`, it has to be used in a `foldhood` expression.

Example: the gradient field Now we show a simple example that will also be used in the security-related experiments of Section 5. It is known as the *gradient* and represents the field of minimum distances from source nodes (Figure 1). This is a fundamental pattern that is used extensively in many applications; for instance, it can be applied to drive escape to security exits by following the shortest paths to them, or to propagate information up to a given area of interest. In ScaFi, it can be implemented as follows:

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

```
def gradient(source: Boolean): Double =
  rep(Double.PositiveInfinity){ distance =>
    mux(source) { 0.0 } {
      foldhood(Double.PositiveInfinity)(Math.min)(nbr{distance}+nbrRange)
    }
  }
```

When 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. The outer `rep` is necessary to keep track of the estimated distance from one round to the next. Note that this gradient algorithm is self-healing, i.e., 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.

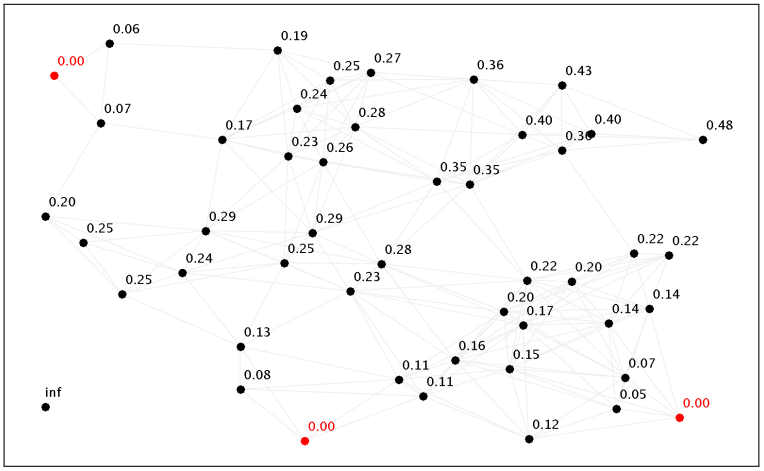


Fig. 1: Snapshot of a stabilised gradient field from a simulation in ScaFi. Devices are represented by bold dots; the red ones denote the sources from which the gradient is computed. The grey lines represent neighbouring links: connected nodes exchange their exports. The nodes are labelled with the string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

3 Aggregate Computing and Security

Aggregate computing systems are susceptible to the security threats that naturally arise from distribution and situatedness. In particular, attacks may target physical devices, the aggregate computing platform, and/or the application logic. At the level of the physical device, sensors may be impaired, networking may be

interrupted, and so on. At the platform level, examples of attacks include Denial of Service (DoS), fake messages, and traffic analysis—just to name a few.

The decentralised nature as well as the peculiar characteristics of adaptivity of the approach make aggregate systems resilient to intermittent or prolonged failure of some nodes—especially for self-stabilising computations like the gradient [20]. 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 crucial is the role a node plays in the application: for example, behaviours building on the gradient algorithm (sec. 2.2) may be ruined if the source nodes fail and the other nodes at some point get rid of the past export[8], though not necessarily in short time, as it also depends on the frequency of round execution 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. 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.

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, it might not be that easy.

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) [15, 6]. 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

[8] The amount of time that neighbour exports are retained depends on the platform configuration for a particular application.

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 the problem at the code-level and propose the use of trust mechanisms to deal with malicious payload.

4 Trust

In the setting of distributed systems, in which it is not possible nor convenient to rely on centralised trusted third parties managing a trustworthiness infrastructure, trust relations rely on direct observations (and potentially recommendations) gathered by every node when interacting with its neighbours. For instance, in trustworthy crowdsourcing and sensor networks, the computation of trust derives from the exchange and aggregation of information disseminated by the participating nodes [8, 23, 9, 16, 3].

Then, trust-based decision-making policies rely typically on the comparison between 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 factors, like, e.g., the initial willingness of the trustor to cooperate with the trustee.

Several trust metrics are based on a Bayesian formulation and, among the various probability distributions proposed in the literature, the beta distribution received particular attention [12, 3, 8, 18]. 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 the observation depends on the context, e.g., 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 of a beta distribution based on such two parameters, as follows:

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

If initially $\alpha = \beta = 0$, then the beta distribution to be used is $\mathrm{Beta}(\alpha+1, \beta+1)$, so that the computed trust expresses a situation of total uncertainty in the absence of any prior interaction between the parties.

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 the neighbours are combined with the parameters discussed above.

4.1 Application to aggregate computing

The proposed idea consists of applying the approach described above to the aggregate computing framework. 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.

Each node maintains locally the pair of parameters (α_i, β_i) for each neighbour i. Their initial value is zero. At each round, every node performs the following operations:

1. The node computes the mean $\bar{x}$ of the values x_i, $1 \leq i \leq N$, read from the N neighbours that have a value to communicate and then, assumed the deviation $\xi_i = x_i - \bar{x}$, computes the mean square deviation:
$$s = \sqrt{\frac{\sum_{i=1}^{N} \xi_i^2}{N}}.$$
2. For each neighbour i, if $|x_i - \bar{x}| > s$ then $\beta_i = \beta_i + 1$, else $\alpha_i = \alpha_i + 1$.
3. For each neighbour i, if $\mathtt{E}(\text{Beta}(\alpha_i + 1, \beta_i + 1)) < tth$ then x_i is discarded.
4. The node computes its local value on the base of the non-discarded x_i.

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.

5 Analysis

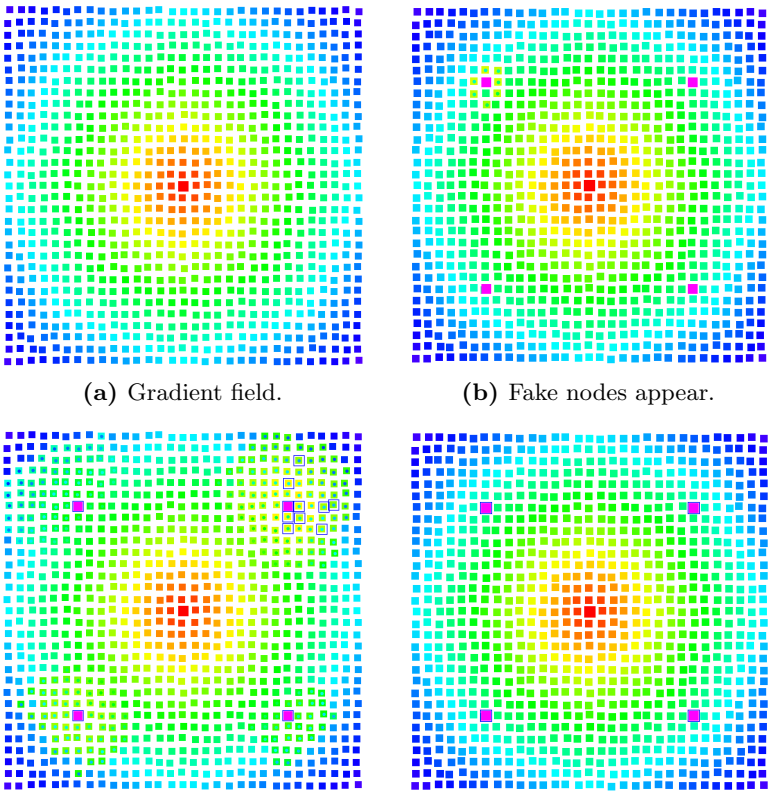
In order to study the trust algorithm proposed in Section 4.1, we have applied it to the case of a gradient computation. We have arranged a set of experiments to exercise the ScaFi implementation of a gradient which makes use of the trust mechanism to detect, mitigate, and even suppress attacks. These experiments take the form of simulations[9] implemented using the Alchemist simulator framework [17] and the corresponding ScaFi incarnation [4].

To put the theory into practice, it should be noted that the effectiveness of this algorithm can be significantly affected by many different factors, such as:

- the threshold s that determines whether to deliver penalties and rewards;
- the trust threshold tth used to actually mistrust 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 aging mechanism;
- the topology of the network; and
- the timing in which events take place.

Moreover, there is a trade-off between prudence and reactivity that should be evaluated on a per-application basis. In this particular study, we have empirically found reasonable values for the aforementioned parameters, and we will

[9] 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 mistrusted and the gradient field is recovered.

Fig. 2: 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 (c), denote the plain gradient field which does not consider fakes (the ideal situation against which the error is calculated). The blue squared contours identify the nodes which are mistrusted by at least one of their neighbours.

mainly focus on the trust score threshold *tth*; a more in-depth analysis of all the interacting factors can be considered as a future work.

The simulation scenario is defined as follows: there are nearly 1000 nodes arranged in a slightly irregular grid; 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 is computed in two flavours: both without considering (G_{ideal}) and taking into account (G_{fake}) fake nodes—

```
class TrustGradient extends AggregateProgram with ScafiAlchemistSupport {
  override def main(): Double = trustedGradient(isSrc, isFake, useTrust)

  def gradient(src: Boolean, fake: Boolean = false, applyTrust: Boolean = false): Double = {
    rep(Double.PositiveInfinity){ distance =>
      def nbrDist = nbr { branch(!fake){ distance }{ fakeValue() } }

      var n, xmean, s = 0.0
      branch(applyTrust){
        n = countHood(nbrDist)
        xmean =  sumHood(nbrDist) / n
        s = Math.sqrt(sumHood{ Math.pow(dist - xmean, 2) } / n)
      }{ /* do nothing */ }

      mux(src) { 0.0 } {
        foldhoodPlus(Double.PositiveInfinity)(Math.min){ // except myself
          val isTrusted = branch(applyTrust){
            trustable(calculateTrust(dist, xmean, s))
          } else { true }
          mux(isTrusted){ nbr{dist}+nbrRange }{ Double.PositiveInfinity }
        }
      }
    }
  }

  def calculateTrust(x: => Double, mean: Double, s: Double): Double = {
    val (nbrId, nbrVal) = nbr(mid(), x)
    val deviation = Math.abs(nbrVal - mean)
    val maxError = Math.max(s, errorLB)

    val obss = rep(MutableField[AlfaBetaHistory]()){ m =>
      val history = m.getOrElse(nbrId, List())
      val obs = if(nbrVal.isFinite){
        if(deviation > maxError) (0.0, 1.0) else (1.0, 0.0)
      } else { (0.0,0.0) }
      m.put(nbrId, (obs :: history).take(observationWindow))
      m
    }.getOrElse(nbrId, List())
    val (a,b) = obss.foldRight((1.0,0.0))((t,u)=>(t._1+u._1, t._2+u._2))
    beta(a+1,b+1)
  }

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

  def trustable(trustValue: Double): Boolean = trustValue >= tth

  // ...
}
```

Fig. 3: Implementation of the trust-based gradient algorithm in ScaFi.

these should represent the optimal and worst-case gradient fields, used as the basis for calculation of the relative errors. We expect the trust-based gradient algorithm G_{trust} to behave exactly as G_{ideal} when there are no fakes and, when they appear, to produce a peak of error that is progressively reduced, tending again to G_{ideal}. Figure 2 depicts the key events and phases of the simulation (in the case of fake nodes at a medium distance from the source), whereas an excerpt of the aggregate program expressing the simulation logic is reported in Figure 3.

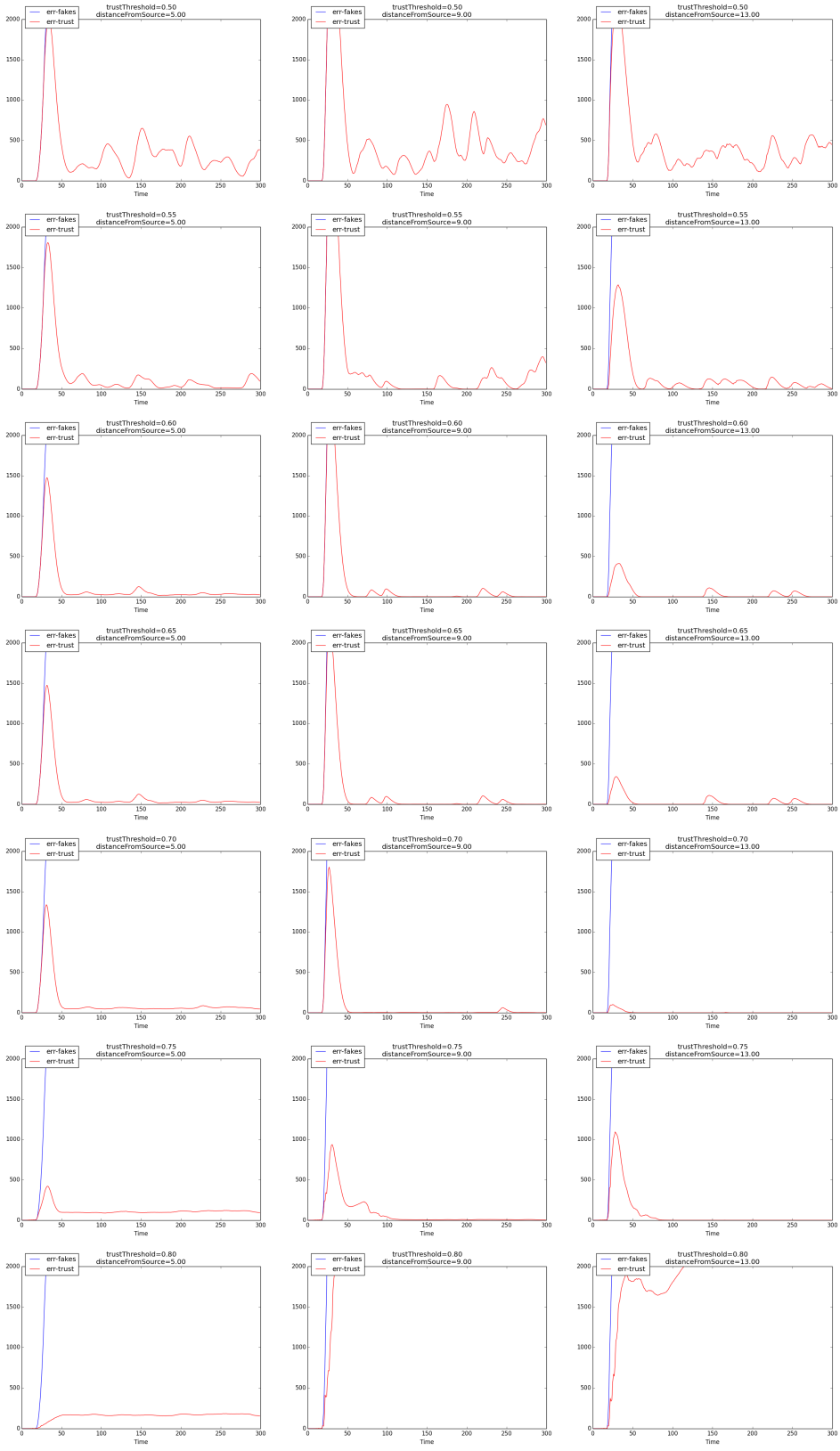


Fig. 4: Sensitivity analysis for the trust threshold.

The results of the sensitivity analysis for the trust threshold tth are reported in Figure 4, which shows, for increasing values of tth (rows), the error committed by the gradient both when it doesn't adopt any form of trust (blue line) and when it does (red line), for three scenarios (columns) of increasing distance of fake nodes from the source. There, it is visible how attitudes that are too permissive (low tth) or too cautious (high tth) can result in greater error or even divergent behaviour. In fact, while the classic `gradient` algorithm completely departs from the ideal situation right after the appearance of fake nodes (simulated with the `fake` flag) at $t = 20$, the use of the trust mechanism enables the system to recover after an initial transitory phase with substantial error. However, the threshold tth on the trust score is crucial in the containment of the deviation: an excessive eagerness to trust neighbours ($tth = 0.50$) leads to accepting too many suspicious messages; up to $tth = 0.70, 0.75$, things improve with the additional severity, but once the circumspection reaches paranoid levels ($tth = 0.80$), too many nodes get distrusted and the system becomes unable to precisely isolate and resolve the problem.

6 Conclusion and Future Work

In this paper, we have for the first time considered security-related aspects in the context of aggregate computing. Though many vulnerabilities are inherent to the distributed nature of aggregate systems, and though the framework is resilient against many temporary sources of failure, the often large attack surface, the suitability of the approach for safety-critical applications (e.g., crowd management and tactical networks), and the fundamental assumption of cooperation on the nodes involved just make the problem worth to be investigated. 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.

Future works include a more in-depth analysis of the factors that affect the trust algorithm applied in this paper, a study of the response of the algorithm when tackling more insidious attacks, as well as a more extensive investigation of the vulnerabilities of the full aggregate computing stack.

References

1. Aldini, A.: Modeling and verification of trust and reputation systems. Journal of Security and Communication Networks 8(16), 2933–2946 (2015)
2. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. IEEE Computer 48(9) (2015)
3. Buchegger, S., Boudec, J.Y.L.: 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)
4. Casadei, R., Pianini, D., Viroli, M.: Simulating large-scale aggregate mass with alchemist and scala. In: Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on. pp. 1495–1504. IEEE (2016)

5. Casadei, R., Viroli, M.: Towards aggregate programming in scala. In: 1st Workshop on Programming Models and Languages for Distributed Computing. p. 5. ACM (2016)
6. Cho, J.H., Swami, A., Chen, I.R.: A survey on trust management for mobile ad hoc networks. Communications Surveys & Tutorials 13(4), 562–583 (2011)
7. Damiani, F., Viroli, M., Beal, J.: A type-sound calculus of computational fields. Science of Computer Programming 117, 17 – 44 (2016)
8. Ganeriwal, S., Balzano, L.K., Srivastava, M.B.: Reputation-based framework for high integrity sensor networks. ACM Trans. Sen. Netw. 4(3), 1–37 (2008)
9. Han, G., Jiang, J., Shu, L., Niu, J., Chao, H.C.: Management and applications of trust in wireless sensor networks: A survey. Journal of Computer and System Sciences 80(3), 602–617 (2014), special Issue on Wireless Network Intrusion
10. Huang, J.: A formal-semantics-based calculus of trust. Internet Computing 14(5), 38–46 (2010)
11. Jøsang, A.: A logic for uncertain probabilities. Int. Journal of Uncertainty, Fuzziness, and Knowledge-Based Systems 9(3), 279–311 (2001)
12. Jøsang, A., Ismail, R.: The beta reputation system. In: 15th Bled Conf. on Electronic Commerce (2002)
13. Li, J., Li, R., Kato, J.: Future trust management framework for mobile ad hoc networks. IEEE Communications Magazine 46(4), 108–114 (2008)
14. Li, Z., Shen, H.: Game-theoretic analysis of cooperation incentives strategies in mobile ad hoc networks. Transactions on Mobile Computing 11(8), 1287–1303 (2012)
15. Marmol, F.G., Perez, G.M.: Security threats scenarios in trust and reputation models for distributed systems. Computers and Security 28(7), 545–556 (2009)
16. Mousa, H., Mokhtar, S.B., Hasan, O., Younes, O., Hadhoud, M., Brunie, L.: Trust management and reputation systems in mobile participatory sensing applications: A survey. Computer Networks 90, 49–73 (2015)
17. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with Alchemist. Journal of Simulation (2013)
18. Priayoheswari, B., Kulothungan, K., Kannan, A.: Beta reputation and direct trust model for secure communication in wireless sensor networks. In: Int. Conf. on Informatics and Analytics. pp. 1–5. ICIA-16, ACM (2016)
19. Trcek, D.: A formal apparatus for modeling trust in computing environments. Mathematical and Computer Modelling 49(1–2), 226–233 (2009)
20. Viroli, M., Beal, J., Damiani, F., Pianini, D.: Efficient engineering of complex self-organising systems by self-stabilising fields. In: Self-Adaptive and Self-Organizing Systems (SASO), IEEE 9th International Conference on. pp. 81–90. IEEE (2015)
21. Viroli, M., Casadei, R., Pianini, D.: On execution platforms for large-scale aggregate computing. In: Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct. pp. 1321–1326. ACM (2016)
22. Viroli, M., Damiani, F., Beal, J.: A calculus of computational fields. In: Canal, C., Villari, M. (eds.) Advances in Service-Oriented and Cloud Computing, Communications in Computer and Information Science, vol. 393, pp. 114–128. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-45364-9_11
23. Yu, Y., Li, K., Zhoub, W., Lib, P.: Trust mechanisms in wireless sensor networks: Attack analysis and countermeasures. Journal of Network and Computer Applications 35(3), 867–880 (2012)