Alma Mater Studiorum · Università di Bologna

Campus di Cesena

Scuola di Ingegneria e Architettura

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Towards Security-Aware Aggregate Computing

Tesi in

Ingegneria dei Sistemi Software Adattativi Complessi

Relatore:

Prof. MIRKO VIROLI

Correlatore:

Prof. ALESSANDRO ALDINI

Presentata da:

GIACOMO MANTANI

Anno Accademico 2015-2016

Sessione III

KEYWORDS

Aggregate Computing

Trust Systems

Field Calculus

Information Security

Distribute Systems

A coloro che...

...mi hanno aiutato nella realizzazione di questo lavoro,
in ordine alfabetico:

Aldini Alessandro
Casadei Roberto
Francia Matteo
Mantani Alessandra
Pianini Danilo
Viroli Mirko

...stimolano la mia passione verso la sicurezza informatica:

i ragazzi del gruppo CeSeNA Security
i miei colleghi di lavoro

...mi stanno sempre vicini:

la mia famiglia
Jessica

I, GIACOMO MANTANI confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Aggregate computing is a paradigm that tries to fully escape the single device abstraction. It handles collective behaviour and interactions between devices for developers.

Thanks to field calculus, aggregate programming can deal with mobile devices deployed in physical space - situated. Every device can move and change position deliberately.

Nowadays aggregate programming does not consider security threads, i.e. a malevolent device that sends corrupted or unexpected data. This is a problem because such algorithms are, above all, deployed in critical systems where human lives could be in danger. It is a priority to look at mitigation and defense mechanisms.

This dissertation presents solutions and finally proposes ideas for new hybrid approaches that use trust system.

Table of Contents

List of Figures

Abbreviations

API Application Programming Interface

DLS Domain-Specific Language

IDS Intrusion Detection System, a device or software application that monitors a network or systems for malicious activity or policy violations

IoT Internet of Things

JSON JavaScript Object Notation

JVM Java Virtual Machine

MANET Mobile Ad-hoc Network

OOP Object Oriented Programming

P2P Peer-to-peer

WSN Wireless Sensors Network

Chapter 1

Introduction

> *The essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context. – John V. Guttag*

In a world of ubiquitous computing, technologies are almost everywhere: bank transfert, communication services, transports and even medical equipment. Devices are mobile and deployed in physical space, situated. Altogether they create what in literature is often called *pervasive continuum*.

The medium is an *amorphous computing medium*, that (Abelson et al. 2000) can be defined as:

> *a system of irregularly placed, asynchronous, locally interacting computing elements.*

Each unit belonging to the medium can *i)* **be faulty**: it is reasonable that it may stop working, *ii)* **be sensitive** to the environment, *iii)* **be able to move** itself around in space, *iv*) **have computing power** and **a memory** system, *v)* **be able to communicate** with some neighbor units.

Pervasive continuum, as every other shared coordination medium must deal with users that could behave inadequately or even with malevolent intents. Information security tries to protect them from illegal uses and threads focusing on confidentiality, integrity and availability of a given system.

For the reasons aforementioned, this work aims at introducing security-aware mechanisms and studying possible solutions in current aggregate programming paradigm. Experiments and solutions proposed focus to a gradient algorithm that evaluate the distance from a source. The gradient is chosen because it is recognised as exemplary model of the computational field.

A gradient algorithm is the *de-facto* standard example and the proposed improvements can be adapted to other scenarios without too much effort. Before starting with the dissertation, in the next section common properties and definitions about collective behaviour follows.

1.1 Collective behavior

Dealing with collective devices' behavior can be more natural and intelligible rather than dealing with each single unit and his interactions with neighbors. The basic application of collective behavior is *wave propagation*. Wave propagation with hop count[1] is evocative of the gradients formed by chemical diffusion. Unit **A** spreads a value to his neighbors forming a chain reaction between nodes. In botanic it is also known as *growing points concept*.

> *A growing point is an activity of a group of neighboring computational particles that can be propagated to an overlapping neighborhood. Growing points can split, die off, or merge with other*

[1] In computer networking, a hop is one portion of the path between source and destination.

> *growing points. As a growing point passes through a neighborhood, it may modify the states of the particles it visits. We can interpret this state modification as the growing point laying down a particular material as it passes. The growing point may be sensitive to particular diffused messages, and in propagating itself, it may exhibit a tropism toward or away from a source, or move in a way that attempts to keep constant the "concentration" of some diffused message. Particles representing particular materials may "secrete" appropriate diffusible messages that attract or repel specific growing points. - (Abelson et al. 2000)*

The characteristics mentioned above recall what (Hewitt & Jong 1984) have stated as properties of every open system:

1. Asynchronous communication for coordination.
2. Ability to deal with large quantities of diverse information.
3. Concurrent unplanned dynamics.
4. Decentralized control, local decisions should be made.
5. Possible inconsistency of information throughout time.
6. Each component needs to keep track only of its own state and how to relate with neighbors.
7. *Principle of locality*:[2] the effect of any event is local.
8. Reliability: failures of individual components should be irrelevant.

[2]In physics, the principle of locality states that an object is only directly influenced by its immediate surroundings.

Accordingly to (Beal & Bachrach 2006), abstraction layers between single devices and the higher level aggregate system are:

- **Global**, higher level that controls the regions'[3] behavior, also called ***computational field*** (Beal & Viroli 2016);
- **Local**, amorphous coordination medium in which the behavior of each node is specific;
- **Discrete**, a network of devices that exchange messages with their neighborhoods.

Devices aggregation can be done through *i)* spatial and time coordination or *ii)* network structure. Specific constructs dictate how data is exchanged and manipulated across regions. Local operations and interactions allow collaboration between individual computational devices.

1.2 Summary of chapters

The outline of the thesis is as follow. **Chapter 2** gives a background on aggregate programming, its paradigm, involved abstractions and properties. **Chapter 3** gives an overview about misbehaviours and the attacks that can occur in aggregate computing systems and, more generally, in distributed system. **Chapter 4** gives a background on trust systems and a description of two of the main algorithms, Eigentrust and Peertrust. **Chapter 5** shows the results of experiments in which attacks are simulated in various scenarios. **Chapter 6** proposes solutions to mitigate attacks investigated in the previous chapter such as *i)* using an external trust system, *ii)* developing filter functions and *iii)* creating an embedded local trust system in each node.

[3] Aggregation of several computational units

Chapter 2

Aggregate Programming

Aggregate programming is a paradigm applied to design, create and maintain complex distributed systems. The aim is to fill up all developers needs when dealing with time, space and an enormous amount of devices interconnected. Such systems are also called the *pervasive continuum* (Zambonelli et al. 2011) for being:

- distributed,
- dense,
- mobile,[1]
- heterogeneous.

It is conform to the following principles useful for distributed programming framework, mentioned by (Beal & Viroli 2016):

1. Have mechanisms for robust coordination *under the hood*.
2. Composing different modules and subsystems must be simple and transparent.

[1]been able to be moved from one place to another

3. Coordination mechanisms could be different between modules, regions and times.

Aggregate programming is currently used in different domains such as wireless sensor networks, crowd safety (Beal et al. 2015), disaster relief operations, construction of resilient enterprise systems and network security. These fields have a common characteristic; they have **computing devices distributed throughout a physical space**. For that reason aggregate programming is always accompanied by the term *spatial computing*.

It can be also applied to networks that are not closely tied to space, such as enterprise service networks for services recovery from failures (Clark et al. 2015).

Nevertheless, it is possible to exploit the aforementioned characteristics in network security. Aggregate programming in network security is inspected by (Paulos et al. 2013).

2.1 Background

Aggregate programming derives from *Field Calculus*, a core set of succinctness constructs which models device behavior and interactions. Thanks to his succinctness it is possible to do mathematical analysis on it. Nevertheless, the field calculus's syntax and semantics are expressive enough to be universal (Beal & Viroli 2016).

In physics, a field is a region in which each point (scalar) is affected by a force (vector) (Britannica n.d.). In this work the terms region and field are used interchangeably. A deeper formalization of **field** notion is described by (McMullin 2002). In mathematics scalars and vectors are indeed quantities

that are used to describe the motion of objects.

The word **computational** means "ability to compute" both of the device by itself and in conjunction with other neighbor devices, exploiting local-to-global interactions. Computation is a pure function over fields (see Figure 2.1): the conception of state is integrated in the one of time.

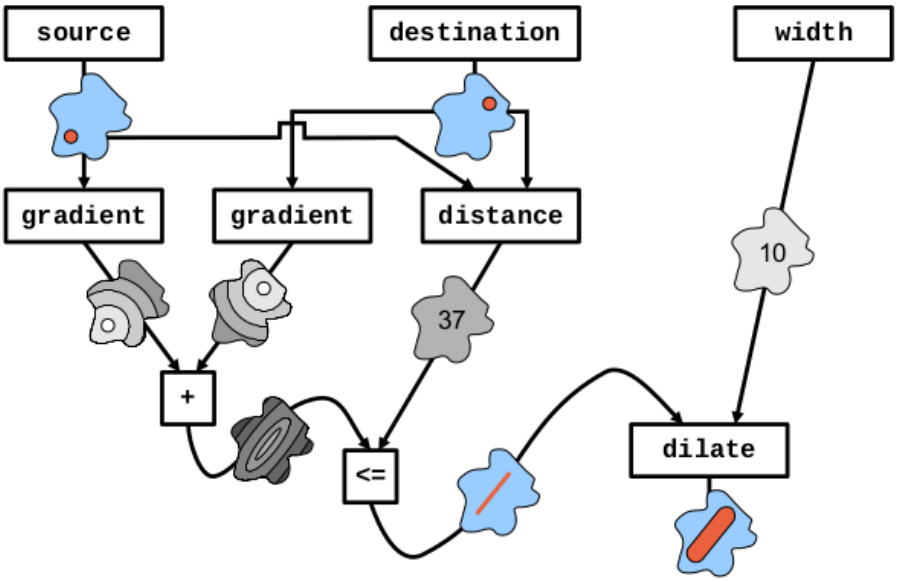

Figure 2.1: How to compute a channel using pure functions

The first computational field model was proposed by (Tokoro 1990). His model aims to solve many large and complex problems of different nature in an open-ended distributed environment, rising the level of abstraction to have a coarse-grained view of the system.

More recent works clarify what a computational field is and how to use it. In particular (Damiani et al. 2016) have presented *i)* a small universal calculus of aggregate-level field calculus, *ii)* core fundamentals constructs and concepts in order to develop sophisticated APIs, both general and domain-specific, for programming distributed systems.

Formally and briefly a field is a space-time structure defined as:

$$\phi : D \mapsto V$$

where:

- D is the events domain, and an event E is a triple $\langle \delta, t, p \rangle$ formed by a device δ "firing" at time t in position p
- V are the field values

See (Damiani et al. 2016) for a more formal and complete syntax.

2.2 Abstractions

It is possible to differentiate aggregate programming in five different layers:[2]

- Application code
- Developer APIs
- Resilient Coordination Operators
- Field Calculus Constructs
- Device Capabilities

Application code is where developers operate through user-friendly developer APIs. APIs encloses the sequence of instructions of resilient coordination operators and field calculus constructs.

A good starting point is the official repository (Pianini 2017) and a forked version with many useful functions collected in a library, *protelis-lang* (Francia [Online; accessed: March 2017]). There are **APIs** that handle:

[2]Higher level of abstraction first

- coordination, such as spreading, shared timer, timer replication, accumulation, tree, graph and more,
- state, such as applying a function while a condition is true, cyclic timer, invoking a function periodically, exponential back-off filter, holding value until timeout and more,
- utilities, such as verifying if a device is on the edge, hood wrapping, min, max and more.

One of the main advantages is the possibility to leave message passing between nodes behind the scene. Behavior results from these APIs and the following coordination operators.

Resilient coordination operators (Beal & Viroli 2014):

- **G**radient-cast: builds a distance-gradient from a source according to a metric. It spreads information across space, potentially further organizing and computing as it proceeds. This operator is a generalization that cove and build accumulated values on it. The G operator may be thought of as executing two tasks *i)* computes of a field of shortest-path distances from a source region and *ii)* computes an accumulation of values along the gradient of the distance field away from the source.
- **C**onverge-cast: collects information distributed across space by accumulating values down the gradient[3] of a potential field. Combining with G it is possible to obtain a general "summary" operator that aggregates the values of a region and then spreads it throughout space.
- **T**ime-decay: summaries information throughout time by decaying accumulated values, i.e. it can be useful in monitoring actions for a given time τ. **2**].

[3]the unique vector field whose dot product with any vector v at each point x is the directional derivative of a scalar function f along v

- **S**parse-choice: creates partitions and selects sparse subsets of nodes in space. It breaks symmetry by exploiting a frequently used self-organization principle, *mutual inhibition*. Devices compete against one another to become local "leaders". For example, it can be used to designate a representative device in a sensor network to act as a collection point and relay to the consumers of the network's sensor data.

Field calculus constructs are formed by:

- **nbr** (= neighborhood), typically application-specific, it is used to obtain values from the nearest devices (i.e., physical proximity, wireless connectivity..);
- **rep** (= repeat), *time evolution*, is a construct for dynamically changing fields, that uses a model in which each device evaluates expressions repeatedly in asynchronous rounds;
- **if**, *domain restriction*, is a space-time branching construct used for example to spread different computation tasks in different regions of the medium. In addition, as depicted in the Figure 2.2 below

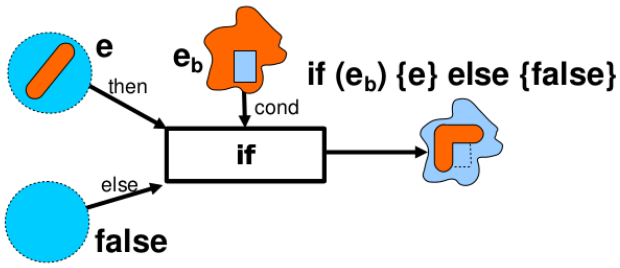

Figure 2.2: If representation

Device capabilities, such as sensors and actuators that collect data from the physical world and supply upper levels.

Recap:

Function	Space	Time
Structure	nbr	rep
Aggregation	C	T
Spreading	G	T
Symmetry breaking	S	random
Restriction	if	if

Higher levels can express:

- Complex spreading, aggregation, decay functions.
- Spatial leader election, partitioning, consensus.
- Distributed *spatio-temporal* sensing and situation recognition.
- Dynamic spreading - eg: code deployment.
- Implicit/explicit device selection of what the code is executed by.
- Creation of sub-groups or collective teams based on the selected code.

2.3 Properties

Each aggregate programming system must have also core properties such as *safety* and *resilience*, guaranteed as much as possible by coordination operators.

(Lamport 1977) has defined safety as a property that - *states that something will not happen.* The basic method of proving safety is to prove consistent behavior of functionalities. (Lamport 1977) also explains a simple formalization of proving safety as:

> If $F : S \to S$ is multivalued function and $D \subset S$ and $F(D) \subset D \implies F^n(D) \subset D \; \forall n > 0$ where F^n is the composition of F with itself n times and F takes the current program state into its next state.

Safety is a characteristic of systems and not of their components. The state of safety in any system is always dynamic.

The property of resilience is the ability to adapt and *self-organize* after changing conditions, such as node failures, node mobility or evolution of the network topology. As (Tanenbaum & Van Steen 2007) said, *resilience incorporates techniques by which one or more processes can fail without seriously disturbing the rest of the system.* Changing the word "processes" with "devices" makes the definition valid for aggregate programming or more generally for all distributed systems.

Self-organization is the spontaneous often seemingly purposeful formation of spatial, temporal, spatio-temporal structures or functions in systems composed of few or many components. The concept of self-organization was discussed by (Ashby 1947) and by (Foerster von H 1987) within his "Cybernetics of second order". It was also discussed in thermodynamics (Nicolis et al. 1977) and (Heylighen & others 2001). A systematic study of self-organization phenomena is performed in the interdisciplinary field of synergetics (Haken 1977) that is concerned with a profound mathematical basis of self-organization as well as with experimental studies of these phenomena. The "Complexity" (Bar-Yam 1997) field is, at least partly, also concerned with self-organization.

Chapter 3

Misbehavior and attacks

> "We cannot choose our external circumstances, but we can always choose how we respond to them" - Epictetus

This chapter starts with the inspection of the meaning of misbehaviour and how it can occur. Next paragraphs are about attacks' taxonomy and definitions. In particular, attacks that are only strictly related with distributed systems and aggregate computing. It is not intended to be a fully observation of all attacks patterns but just an overview of plausible strategies used with malevolent intent.

3.1 Misbehaviours

> *A distributed system is constructed from a set of relatively independent components that form a unified, but geographically and functionally diverse entity. Nevertheless, distributed systems remain difficult to design, build, and maintain.* - Rob Pike, Dennis M. Ritchie

In a distributed system each independent component can obviously leave the system or fault and stop working.

Misbehavior is a generic term to say that something does not operate as it should. Node misbehaviour can occur:

- by *i)* intentionally (malevolent) or *ii)* accidentally performing an action that is not allowed.
- by failing to perform an action due to lack of data and/or resources or hardware faults.

This research takes a look only to misbehaviours that does not affect not faulty nodes, directly or indirectly. Note that nodes are able to communicate with each other only through message passing, the only medium disposable.

During message passing there are three ways in which nodes can influence one another:

- by not receiving a message.
- by receiving a malformed message.
- by receiving a well formed message but with bad data.

The last two are slightly different; the first one is about the structure and the syntax of the message, whereas the latter is about the semantics, the meaning of the message.

Common distributed system properties (Gray 1986), such as availability, reliability and fault-tolerance are already taken into account by aggregate programming. Unfortunately these properties do not often take into account problems that occurs with intentionally produced misbehaviours.

3.2 Attacks

Misbehavior, as seen in the previous section is a generic term. It is possible to differentiate a much broader range of intentional misbehaviours, commonly called attacks. Attacks can leverage fundamental system properties, come up with really smart ideas and be really unpredictable. In this section an overview and a taxonomy of plausible attacks in a typical WSN and aggregate programming scenario is presented.

Each time an attack is described, it is important to keep in mind which is the purpose and who is the target. From a security perspective it is a good practice to think about the knowledge required and what is the amount of time and/or resources needed by an attacker to accomplish his malevolent intents.

An attack can be done by a single or a group.

3.2.1 From a single entity

Single malevolent entities attacks are:

- *Corruption*, altering data randomly or forged.
- *Selective misbehaving*, exchange valuable and good information to someone and erroneous or bad to others.
- *On-off*, an entity may behave badly and well alternatively.

3.2.2 From multiple cooperating entities

In addition to the single and isolated malevolent entity, there could be cooperating ones. Attack efficiency is obviously greater and the system can

be drastically negatively affected. In literature, cooperating attacks are also called *collusions*, and they are:

- *Driving down*: malevolent entities outnumber the good one, dictating the system.
- *Malicious collectives with camouflage*: malevolent groups collectively send positive information about a malicious agent. This attack is also called *ballot stuffing* and it is well-represented by (Mármol & Pérez 2009).
- *Malicious spies*: a subtle *malevolent collectives with comouflage* where entities behave always correctly in order to gain trust from good peers and from this "status quo" they give positive feedback to other malevolent entities in the network.

3.2.3 Attacks properties

The previously mentioned attacks' taxonomies, were created accordingly to the following properties:

- **Attack intent**: An enemy may have several different goals when trying to subvert a system.
- **Targets**: Security threats can focus their efforts on a subset of entities belonging to the system, on (non-)specific individual targets.
- **Required knowledge** : The amount of information that has to be gathered or collected from the system in order to effectively perform an attack Thus, some threats will require a comprehensive knowledge about the whole system or about some particular entities, while some other threats will work properly with a small knowledge.

- **Cost**: The less expensive an attack is, the more beneficial is its application. The cost of running an attack is not necessarily pecuniary, but it can be also measured in terms of resources or time.
- **Algorithm dependence**: Some security threats take advantage of a specific algorithm or of the model vulnerability and exploits it in order to create a great damage to the system. Other attacks are more generic and, consequently, applicable in a wider set of scenarios or environments.
- **Detectability**: The later an attack is detected, the greater might be the damage. That is the reason why most of the threats try not to induce suspicion as much as possible, i.e., they do not cause drastic changes in the system, but they rather make slight ones. In some way, the ability to detect an attack or threat is an evaluation of its resilience and effectiveness. If the collaboration between attackers increases, as well as their gathered knowledge about the system, it is more difficult to detect or prevent them.

3.3 Possible mitigations

In order to catch the aforementioned misbehaviours and attacks, a refactoring of entities/nodes behavior' is needed within the standard framework used. Some reasonable mitigations could be:

- Injecting memories in each node concerning previous received data in order to detect not common messages. Unsupervised learning mechanism can be used.
- Adding filters that drop not conformed messages.
- Using a trust system and reputation model, so that a message from an un-trusted or unknown entity could be dropped or used with caution.

Chapter 4

Trust Systems

Trust research range over different domains and topics such as sociology, economics, philosophy, psychology, organizational management, autonomic computing, communications and networking (Cho et al. 2011). It is a real multidisciplinary concept with a term not yet completely formalized. According to Cambridge Dictionary (Dictionary [Online; accessed: March 2017]), trust is defined as *"to believe that someone is good and honest and will not harm you, or that something is safe and reliable"*.

Trust can be perceived as someone/something's **reliability** or as a willing dependence, **decision**.

4.1 Background

Trust is a one-directional relationship between two peers that can be called **trustor** and **trustee**. The trustor is the entity that has the ability to make assessments and decisions based on the information received and on its past experience with the trustee.

Trust can be differentiated into two macro areas *i)* **IT security** and *ii)* **soft-security** (Jøsang 2007). The latter deals with human interactions and it relates to psychological factors. For this reason it is not considered in this study, which aims at pragmatic and practical approaches.

Reputation is often related to trust but these two concepts differ. The former is a collective feedback-delivered quantity which is shared by the whole community, The latter is a *personal* and *subjective* opinion that can be seen as a score based only on direct experience.

A trust metric must has the following characteristics (Cho et al. 2011):

- It should be established referring to potential risks.
- It should be context-dependent.
- It should be based on its own interest.
- It should be easily adaptable, dynamic and constantly updating.
- It should mirror the system reliability.

The term trust and **trustworthiness** have different meanings. Trust is a belief and trustworthiness is the actual probability that varies from zero (distrusted) to one (trusted). The perceived or estimated trustworthiness of a potential cooperation partner is the basis for the trustor's decision to whether or not to cooperate. Figure 4.1 shows some important relationships.

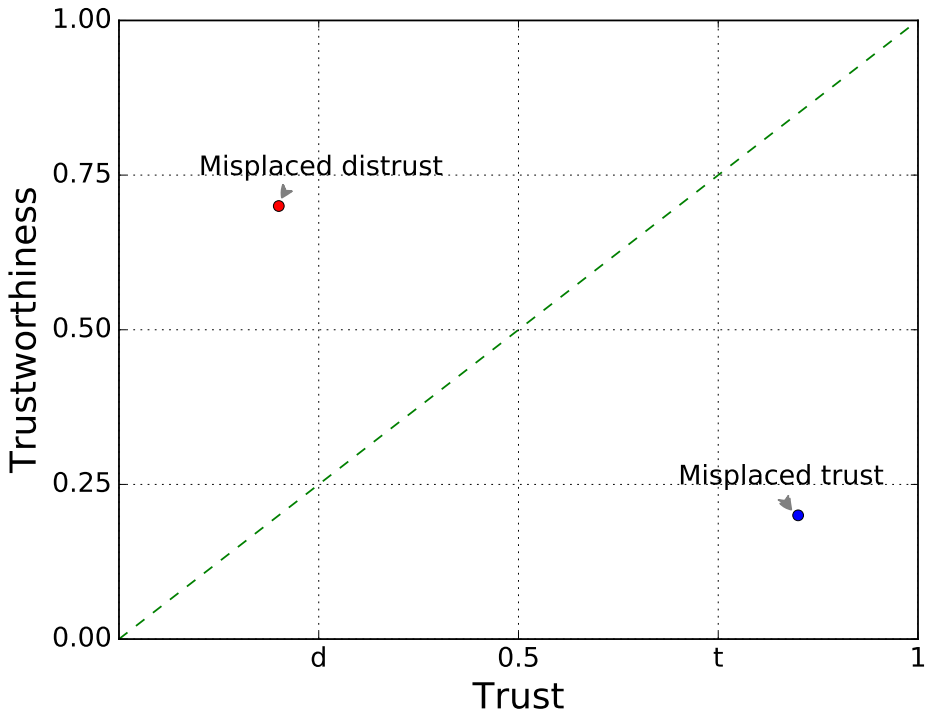

Figure 4.1: Trust levels: The horizontal axis is the subjective probability, i.e. the level of trust, whereas the vertical axis is the objective probability, i.e. the level of trustworthiness. The t on the horizontal axis marks the trust threshold, so there is trust whenever the subjective probability is greater than t.

A crucial observation now is that a trust level $\geq t$ (resp. $\leq d$) is not enough for engaging in cooperation (Solhaug et al. 2007).

Obviously, trust per se is related with risk calculation and the aforementioned probabilities are important metrics to cope with (Jøsang & Presti 2004). (Solhaug et al. 2007) conclude that trust is generally neither proportional nor inversely proportional to risk.

There are a multitude of applications of trust system such as EigenTrust (see Section 4.2.1), PeerTrust (see Section 4.2.2), BTRM-WSN and PowerTrust. Most of them are surveyed by (Mármol & Pérez 2009) with a notable description and worth comparison.

(Mármol & Pérez 2009) try to enumerate the fundamental steps needed in a typical trust system as follows:

1. Collecting information.
2. Aggregating all the received information properly and somehow computing a score for every peer in the network.
3. Selecting the most trustworthy messages and assessing a posteriori the satisfaction updating the score.
4. (Optional) A last step, "punishing[1] or rewarding" can be carried out, adjusting consequently the global trust (or reputation).

Once misbehaving nodes are detected, their neighbors can use trust information to avoid cooperation with them as if they are an obstacle.

4.2 Trust Systems overview

It is worth to mention at least two of the main trust algorithms before proposing a solution. In the following sections a brief summary of two trust algorithms - *i)* EigenTrust and *ii)* PeerTrust - follows:

4.2.1 EigenTrust

EigenTrust Algorithm (Kamvar et al. 2003) was proposed in 2003. The algorithm has been incrementally developed with additional features in each

[1]Bucchegger and Le Boudec think that liars, node that report inaccurate testimonials, should not be shamed or punished. The reason is that punishing these messages discourage honest reporting of observed misbehavior. Since there are always at some point a node that is bound to be the first witness of another node that misbehave, thus starting to deviate from public opinion could be punished wrongly.

revision. The basic EigenTrust algorithm has a simple centralized reputation calculation strategy, while the advances include distributed, transitive and secured strategies for global calculations. An overview of the distributed strategy of the algorithm follows. (Kamvar et al. 2003) have a deeper description.

The next example can be easily adapted to other contexts such as aggregate programming devices or nodes.

Consider a P2P system consisting of n peers. Each time peer i exchanges information with peer j, it rates the communication as $sat(i, j)$ if positive, and $unsat(i, j)$ if negative, and keeps a record for the number of each. Then the trust value s_{ij} is defined as:

$$s_{ij} = sat(i, j) - unsat(i, j) \tag{4.1}$$

In order to aggregate local trust values, it is necessary to normalize them in some manner. Otherwise, malevolent peers can assign arbitrarily high local trust values to other malevolent peers, and arbitrarily low local trust values to good peers, subverting the system easily. The trust value s_{ij} is normalized as follows:

$$c_{ij} = \frac{max(s_{ij}, 0)}{\sum_j max(s_{ij}, 0)} \tag{4.2}$$

Normalized local trust value $\langle c_{ij} \rangle$ ensure that all values will be between 0 and 1.

Usually, there are some peers that are known to be trustworthy in any P2P system, so they are identified at an early stage of the system life as a set of pre-trusted peers, P. This is especially important for inactive peers or those

who recently joined the system, as they do not trust any peer. Thus, the trust value is redefined as:

$$c_{ij} = \begin{cases} \frac{max(s_{ij},0)}{\sum_j max(s_{ij},0)} & \text{if } \sum_j max(s_{ij},0) \neq 0 \\ p_i & \text{otherwise} \end{cases} \tag{4.3}$$

where

$$p_i = \begin{cases} \frac{1}{|p|} & \text{if } i \in P \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

Peer i's global reputation is given by the local trust values given to it by other peers, weighted by the global reputation of the assigning peers. Let C be the matrix $[c_{ij}]$ and $\overrightarrow{c_i}$ a vector defined as follows:

$$C = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,j} & \cdots & c_{1,m} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,j} & \cdots & c_{2,m} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ c_{i,1} & c_{i,2} & \cdots & c_{i,j} & \cdots & c_{i,m} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,j} & \cdots & c_{m,m} \end{pmatrix}, \overrightarrow{c_i} = \begin{pmatrix} c_{i,1} \\ c_{i,1} \\ \vdots \\ c_{i,j} \\ \vdots \\ c_{i,m} \end{pmatrix} \tag{4.5}$$

Having this, t_{ik} represents the trust that peer i places in peer k based on asking his friends, and defined as:

$$t_{ik} = \sum_j c_{ij} c_{jk} \tag{4.6}$$

In matrix notation, $\overrightarrow{t_i}$ is the vector containing the values t_{ik}.

$$\overrightarrow{t_i} = C^T \overrightarrow{c_i} = \left(\sum_{j=1}^{n} c_{ij} c_{j1}, \ldots, \sum_{j=1}^{n} c_{ij} c_{jk}, \ldots, \sum_{j=1}^{n} c_{ij} c_{jm} \right) \tag{4.7}$$

By querying his friends' friends, peer i gets a wider view of peer's k reputation, that is:

$$\overrightarrow{t_i} = (C^T)^2 \overrightarrow{c_i} \tag{4.8}$$

Going on in this way, after a large enough number m of queries, peer i will get the same eigenvector $\overrightarrow{t_i} = (C^T)^m \overrightarrow{c_i}$, as every other peer in the system. Additionally, authors propose more sophisticated ways of computing this eigenvector based on pre-trusted peers.

(Kamvar et al. 2003) consider that a peer who is honest providing something is also likely to be honest in reporting its local trust values, which is not necessarily always true.

There is also a distributed version where all peers in the network cooperate to compute and store the global trust vector in order to reduce computation, storage and message overhead for each peer.

4.2.2 PeerTrust

PeerTrust (Xiong & Liu 2004) is a trust and reputation model that combines several important aspects related to the management of trust and reputation in distributed systems, such as:

- The feedback a peer receives from other peers
- The total number of transactions of a peer
- The credibility of the recommendations given by a peer

- The transaction context factor
- The community context factor

This accurate aggregation is performed through the following expression, representing the trust value of peer u:

$$T(u) = \alpha \sum_{i=1}^{I(u)} S(u,i)CR(p(u,i))TF(u,i) + \beta \times CF(u) \tag{1}$$

where

- $I(u)$ denotes the total number of transactions performed by peer u with all other peers;
- $p(u,i)$ denotes the other participating peer in peer u^{ith}'s transaction;
- $S(u,i)$ denotes the normalized amount of satisfaction peer u receives from $p(u,i)$ in its i^{ith} transaction;
- $CR(v)$ denotes the credibility of the feedback submitted by v;
- $TF(u,i)$ denotes the context factor of the adaptive transaction for peer u^{ith}'s transaction;
- and $CF(u)$ denotes the context factor of the adaptive community for peer u.

On the other hand, the credibility of v from w's point of view, is computed as:

$$Cr(p(u,i)) = \frac{Sim(p(u,i),w)}{\sum_{j=1}^{I(u)Sim(p(u,j),w)}} \tag{2}$$

where

$$Sim(v,w) = 1 - \sqrt{\sum_{x \in IJS(v,w)} \left(\frac{\sum_{i=1}^{I(x,v)} S(x,i)}{I(x,v)} - \frac{\sum_{i=1}^{I(x,w)} S(x,i)}{I(x,w)} \right)^2 \Big/ |IJS(v,w)|} \tag{3}$$

and respectively:

- $I(u,v)$ denotes the total number of transactions performed by peer u with peer v;
- $IS(v)$ denotes the set of peers that have interacted with peer v;
- $IJS(v,w)$ denotes the common set of peers that have interacted with both peer v and w, computed as $IS(v) \cap IS(w)$.

Additionally this model introduces a trust-based peer selection scheme.

A simple rule for peer w to decide whether to have an interaction with peer u or not could be $T(u) > T_{threshold}(w)$, where the value of $T_{threshold}(w)$ depends on several factors such as the importance of the transaction, or the disposition of w to trust unknown peers or not, among many others.

Chapter 5

Experiments

> *The road to learning by precept is long, but by example short and effective.* Lucius Annaeus Seneca

This chapter adds together the knowledge gathered from the previous chapters and tries to to find out how to develop solutions by following the scientific method. Solutions tackle some security issues in a common aggregate programming algorithm: the gradient.

A **gradient** field is often associated with the function `distance-to`. Function `distance-to` takes as its input a source field and returns a new field that maps each node to the estimated distance from the nearest source device.

A **source** is a field holding specific values that differentiate it from other fields. Similarly in these experiments malevolent nodes have a value that differentiates them from possible other sources and devices.

It is important to be aware that in real scenarios malevolent nodes could use fake identities. It is not possible to identify good peers from bad ones before adding an authentication algorithm. Proposed solutions must somehow differentiate and identify "fake" nodes. Nevertheless, algorithms' mentioned

before checks and filters which values each peer sends to his neighbours.

The scenario tested is on a grid-like network in which malevolent nodes send erroneous values such as:

1. Camouflage as a real source:
    - with a different distance from the real one;
    - with an incremental number of nodes in it;
    - in groups/crowds.
2. Send corrupted values to their neighbours.

The first sections start with a brief overview of the used languages:

- Protelis Language (Par. 5.1)
- Scala (Par. 5.3)

and technologies:

- Alchemist Simulator (Par. 5.2)
- scafi (Par. 5.3)
- SciPy (Par. 5.4)

The next following sections are about the experiments and simulations, in which malevolent area change:

- distance from source (Par. 5.5.1)
- size (Par. 5.5.2)
- number of (Par. 5.5.3)
- sent values (Par. 5.5.4)

The last sections are about solutions (Par. 5.6):

- Solution one: External Trust System (Par. 5.6.1)
- Proposed Solution two: Filter Functions (Par. 5.6.2)
- Proposed Solution three: Local Trust system (Par. 5.6.3)

5.1 Protelis Language

The goal of the Protelis language is to make resilient networked systems easy to build for complex and heterogeneous networks as for single machines and cloud systems. This is accomplished by separating the different tasks and making some of the hard and subtle parts automatic and implicit. A few of the key design decisions behind Protelis are:

- It is a language because there are a lot of subtle and easy ways to break a distributed system. Creating a language, rather than just a library, lets complexity be handled implicitly, so there is no opportunity to make mistakes.
- It is hosted in and integrated with Java, an advantage thanks to its large pre-existing ecosystem and libraries.
- It looks as much practical as Java in order to make it easier to learn and adopt.
- It ensures safe and resilient composition because its core is field calculus, the theoretical model discussed in 2.1.

Protelis emerged from the synthesis of several prior projects:

- **Proto**, an aggregate programming language created by Jacob Beal and Jonathan Bachrach.

- **Field calculus**, a distillation of aggregate programming models by Mirko Viroli, Ferruccio Damiani, and Jacob Beal
- The **Alchemist Simulator** project, led by Danilo Pianini and Mirko Viroli.

5.2 Getting a visual representation: Alchemist

Alchemist is an extensible "meta-simulator", inspired by stochastic chemistry (i.e. kinetic Monte Carlo) and tailored to pervasive computing and distributed systems. It provides a flexible meta-model, on which developers should bind their own abstractions. An abstraction is also called incarnation or extension. In this research the Protelis language[1] incarnation is used in order to use field calculus constructs.

Alchemist is available on Maven Central. It is possible to import all the components using the `it.unibo.alchemist:alchemist` groovy artifact. If you do not need the whole Alchemist machinery but just a sub-part of it, you can restrict the set of imported artifacts by using as dependencies the modules you are actually in need of.

Using Gradle build system[2] it is only required to specify in the Groovy[3] script named `build.gradle` which alchemist version to use:

```
compile 'it.unibo.alchemist:alchemist:ALCHEMIST_VERSION'
```

Groovy is an optionally typed and dynamic language, with static-typing and

[1] protelis.github.io ([Online; accessed: March 2017])
[2] gradle.org ([Online; accessed: March 2017])
[3] www.groovy-lang.org ([Online; accessed: March 2017])

static compilation capabilities. An example of build script is structured as follows:

```
1
2  dependencies {
3      compile ( "it.unibo.alchemist:alchemist:$alchemistVersion" )
4  }
5
6  task runAlchemist(type: JavaExec) {
7      classpath = sourceSets.main.runtimeClasspath
8      classpath "src/main/protelis"
9      classpath "src/main/java"
10     maxHeapSize = "8g"
11     main = "it.unibo.alchemist.Alchemist"
12     args(
13         "-y", "src/main/yaml/${simulation}.yml",
14         "-g", "src/main/effects/${simulation}.aes",
15         "-t", "500",
16         "-e", "data/exportedData"
17     )
18     dependsOn(compileJava)
19 }
20
21 defaultTasks("runAlchemist")
22
```

It is possible to get ready for the first simulation using the Protelis incarnation only by cloning and running Protelis-Sandox repository (Francia [Online; accessed: March 2017]).

This work has used Alchemist only to have a visual representation of how an attack scenario looks like. Simulations have been done using scafi, as sections below will further inspect. Alchemist network topology and scafi match.

Experiment sources can be found at GitHub (github.com/jak3/Protelis-Sandbox [Online; accessed: March 2017]).

5.3 Data extraction and simulations: scafi

`scafi` is an aggregate programming framework for Scala. It provides:

1. a Scala-internal DSL for expressing aggregate computations and field calculus semantics in a correct and complete way;
2. a distributed platform developed over the Actor Model of Akka toolkit[4], supporting the configuration and execution of aggregate systems;
3. flexible API.

`Scala` is a general-purpose programming language with the following main characteristics:

- it runs on the JVM and integrates with the Java ecosystem seamlessly;
- it is a pure OOP: every value is an object and every operation is a method call;
- unlike Java, Scala has many features of functional programming languages that integrate smoothly with the object-oriented paradigm;
- has an advanced type system supporting algebraic data types, covariance and contravariance, higher-order types, and anonymous types;
- is designed to be a "scalable" language, by keeping things simple while complexity grows;
- has a powerful and expressive static type system with type inference.

`scafi` provides a reasonably fast Virtual Machine for the scafi DSL and it comes with a basic simulator that allows execution of aggregate algorithms locally. In this research it is used to simulate attacks scenarios and data extraction for statistic purpose and plotting.

[4]akka.io ([Online; accessed: March 2017])

5.4 Plotting: matplotlib

Plotting has been done using matplotlib from `SciPy`. `SciPy` is a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, these are some of the core packages:

- NumPy: Base N-dimensional array package.
- SciPy: Fundamental library for scientific computing.
- Matplotlib: Comprehensive 2D Plotting.
- IPython: Enhanced Interactive Console.
- Sympy: Symbolic mathematics.
- pandas: Data structures & analysis.

To get a glimpse of its simplicity, the source code used to plot error functions follow:

```
1
2  def plotError(metadata, errors):
3      plt.grid(True)
4      plt.title(metadata['title'])
5      plt.xlabel(metadata['lx'])
6      plt.ylabel(metadata['ly'])
7      plt.margins(x=0.04, y=0.04)
8      plt.tight_layout()
9      plt.xticks(metadata['x'])
10
11     x = metadata['x']
12     z = np.polyfit(x, errors, 3)
13     f = np.poly1d(z)
14     # Return evenly spaced numbers over a specified interval
15     x_new = np.linspace(x[0], x[-1], 100)
16     y_new = f(x_new)
17     plt.plot(x, errors, '--or', x_new, y_new)
18
```

5.5 Simulations

As previously mentioned, the different types of malevolent behaviour that are considered are:

1. Fake source at different distance from the real one (Section 5.5.1).
2. Fake source of different size (also referred as magnitude) (Section 5.5.2).
3. Number of fake sources (Section 5.5.3).
4. Degree of maliciousness of the fake source: using lower values in a gradient algorithm has a greater impact on the system (Section 5.5.4).

Each section has:

- a visual representation that helps to grasp the scenario, thanks to Alchemist;
- a snipped scala code used to run the simulation and export data;
- a plot, useful as a recap and forecast of other similar scenarios.

Error function is calculated as follows:

$$\epsilon = |\mu_{healthy} - \mu_{misbehaviour}|$$

5.5.1 Distance

Alchemist Screenshots

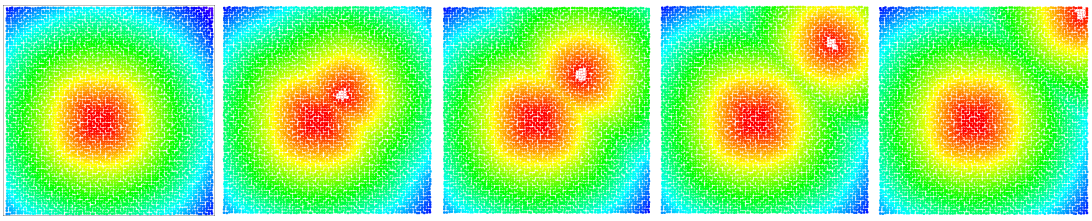

Figure 5.a

In (Figure 5.a)[5] the first image from left shows an healthy gradient in which source field is at the center of the red area. Subsequently a malevolent field was added and nodes were highlighted with white. From left to right, it is possible to see the damage that a fake source produces at different distances. The more each simulated gradient colors differ from the first picture, the greater the network was influenced.

scafi code snippet

```
1
2  object DemoSecurity extends AggregateProgram with MyLib {
3
4    def isSource = sense[Boolean]("source")
5    def isFake = sense[Boolean]("fake")
6
7    def gradient(source: Boolean)(expr:Double): Double = {
8      rep(Double.MaxValue) {
9        distance =>
10         mux(isFake) { expr }
11         { mux(isSource) { 0.0 } {
12             minHood {
13               nbr { distance } + nbrRange()
14             }
15           }
16         }
17     }
```

[5] All images are captured after the gradient was stabilized.

```
18    }
19
20    def main = gradient(isSource){ 0.0 }
21  }
22
23  object DemoSequenceLauncher extends App {
24    for (a <- 0 to 15) {
25      val net = simulatorFactory.gridLike(n = 15, m = 15,
26                                          stepx = 1, stepy = 1,
27                                          eps = 0.0, rng = 1.1)
28
29      net.addSensor(name = "source", value = false)
30      net.chgSensorValue(name = "source",ids = Set(3),value=true)
31
32      net.addSensor(name = "fake", value = false)
33      net.chgSensorValue(name = "fake",ids = Set(a),value = true)
34
35      net.executeMany(node = DemoSecurity, size = 10000,
36                      (n,i) => {})
37    }
38  }
```

Chunk of an output file:

```
0.0  1.0  1.0  0.0  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0
1.0  2.0  2.0  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0
2.0  3.0  3.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0
3.0  4.0  4.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0 11.0
4.0  5.0  5.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0 11.0 12.0
5.0  6.0  6.0  5.0  6.0  7.0  8.0  9.0  10.0 11.0 12.0 13.0
6.0  7.0  7.0  6.0  7.0  8.0  9.0  10.0 11.0 12.0 13.0 14.0
7.0  8.0  8.0  7.0  8.0  9.0  10.0 11.0 12.0 13.0 14.0 15.0
8.0  9.0  9.0  8.0  9.0  10.0 11.0 12.0 13.0 14.0 15.0 16.0
```

It is also possible to use just Alchemist instead of using both. Source code to simulate a fake source at a different distance from the real one can be done with the following Protelis file:

```
1
2  module distance
3
4  let res = rep (d <- 100) {
5    mux(env.has("fake")) { 0 } else{
6      mux (env.has("source")) { 0 } else {
7        minHood(nbr(d) + self.nbrRange())
8  }}};
```

```
9  env.put("distanceTo", res);
10 res
11
```

and Yaml configuration:

```
1
2  incarnation: protelis
3
4  network-model:
5    type: EuclideanDistance
6    parameters: [0.5]
7
8  pools:
9    - pool: &distance
10     - time-distribution: 1
11       program: distance
12     - time-distribution: null
13       program: send
14
15 displacements:
16   - in:
17       type: Grid
18       parameters: [-10, -10, 10, 10, 0.25, 0.25, 0.1, 0.1]
19     contents:
20       - in:
21           type: Rectangle
22           parameters: [5, 5, 5, 5]
23         molecule: viri
24         concentration: "Fake"
25       - in:
26           type: Rectangle
27           parameters: [-2, -2, 2, 2]
28         molecule: source
29         concentration: true
30     programs:
31       - *distance
```

Simulations results

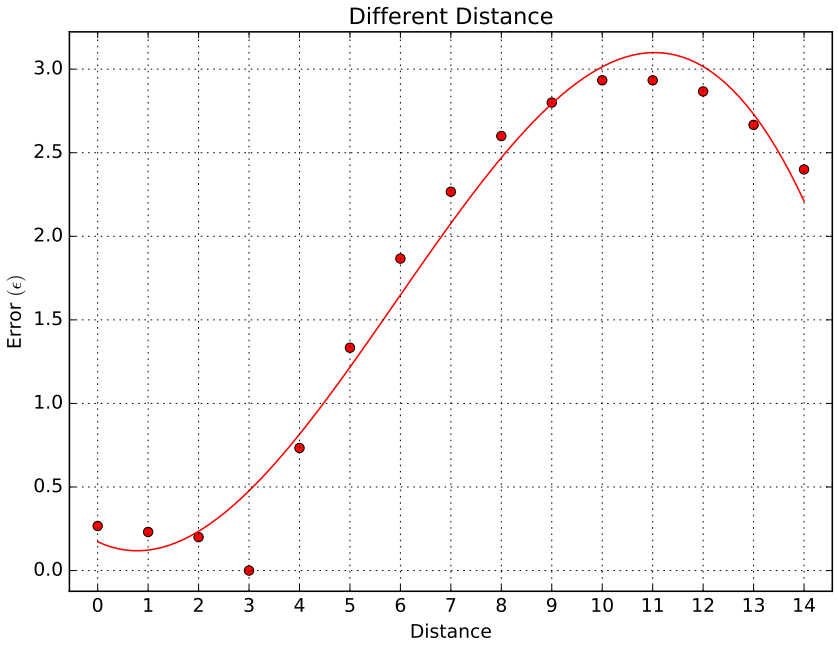

Figure 5.1

At a first look, it is possible to think that the more the fake source is near to the real one the greater the damage will be. However Figure 5.1 proves otherwise. It is rational because, in a gradient based on a `distance-to` algorithm, malevolent nodes near a source have values with a lower δ. Far away from the source, nodes have a high internal value and they are clearly more influenced by a neighbor that sends 0.0 as distance value. Error increases with an exponential growth until the fake source exits the gradient.

5.5.2 Magnitude

Alchemist Screenshots

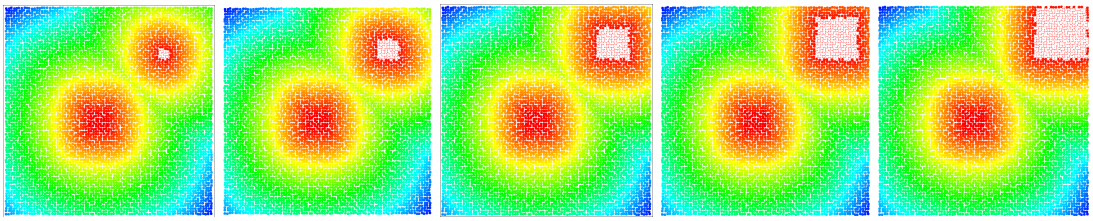

Figure 5.b

A healthy gradient can be seen in the previous section (Section 5.5.1). Malevolent nodes are represented in white[6], firstly fewer than the real ones and gradually more.

scafi code snipped

```
1
2  object DemoSecurity extends AggregateProgram with MyLib {
3
4    def isSource = sense[Boolean]("source")
5    def isFake = sense[Boolean]("fake")
6
7    def radius(x: Double, r: Int): Double = {
8      if (x < r) return 0
9      else return x
10   }
11
12   def gradient(source: Boolean)(expr:Double): Double = {
13     rep(Double.MaxValue) {
14       distance =>
15         mux(isFake) { expr }
16         { mux(isSource) { 0.0 } {
17             minHood {
18               nbr { distance } + nbrRange() }
19           }
20         }
21     }
22   }
23   def main = gradient(isSource){
```

[6] All images are captured after the gradient was stabilized.

```
24                G[Double](isSource, 0,
25                          (x: Double) => radius(x, 2), 1)
26              }
27
28  }
29
30  object DemoSequenceLauncher extends App {
31
32    def Magnitude() {
33      val boss = 25*5+15
34
35      def getBad(c: Int, r:Int, size: Int): Set[Int] = {
36        var init = Set[Int]()
37        for(i <- 0 to r) {
38          for (k <- 0 to r) {
39            init += c + (size * i + k)
40          }
41        }
42        return init
43      }
44
45      val badguys = Array(Set(boss), getBad(boss-26, 2, 25),
46                    getBad(boss-(26*2), 3, 25),
47                    getBad(boss-(26*3), 4, 25))
48
49      for(magnitude <- 0 to 8){
50        val net = simulatorFactory.gridLike(n = 15, m = 15,
51                                            stepx = 1, stepy = 1,
52                                            eps = 0.0, rng = 1.1)
53
54        net.addSensor(name = "source", value = false)
55        net.chgSensorValue(name = "source", ids = Set(25*3+5),
56                           value = true)
57        net.addSensor(name = "fake", value = false)
58        net.chgSensorValue(name = "fake",
59                           ids = getBad(boss-26,magnitude, 25),
60                           value = true)
61        net.executeMany(node = DemoSecurity, size = 100000,
62                        (n,i) => {})
63      }
64  }
```

Plotting results

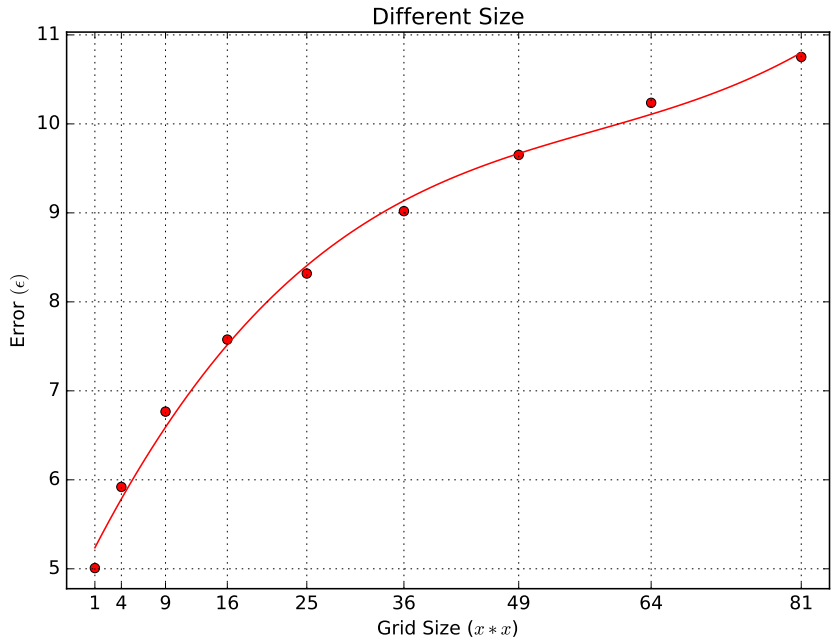

Figure 5.2

Intuitively the greater the malevolent area size is, the more the system is influenced by it.

5.5.3 Crowds

Alchemist Screenshots

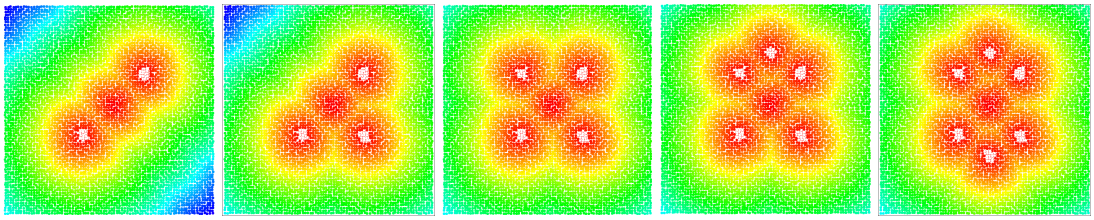

Figure 5.c

Crowds of malevolent areas surround the real source, starting from two fake independent zones[7] to end with six.

scafi code snipped

```
1
2  object DemoSecurity extends AggregateProgram with MyLib {
3
4    def isSource = sense[Boolean](”source”)
5    def isFake = sense[Boolean](”fake”)
6
7    def gradient(source: Boolean)(expr:Double): Double = {
8      rep(Double.MaxValue) {
9        distance =>
10         mux(isFake) { expr }
11         { mux(isSource) { 0.0 } {
12             minHood {
13               nbr { distance } + nbrRange()
14             }
15           }
16         }
17     }
18   }
19
20   def main = gradient(isSource){ 0.0 }
21
22 }
23
24 object DemoSequenceLauncher extends App {
25   def Crowd() {
```

[7] All images are captured after the gradient was stabilized.

```
26      val init = 25*25/2
27      val pos = Set(init+10, init-10, init-25*3,
28                    init+25*3, init-20*3, init+20*3)
29
30      for(b <- 1 to 7){
31        val net = simulatorFactory.gridLike(n = 15, m = 15,
32                                            stepx = 1, stepy = 1,
33                                            eps = 0.0, rng = 1.1)
34
35        net.addSensor(name = "source", value = false)
36        net.chgSensorValue(name = "source", ids = Set(init),
37                           value = true)
38        net.addSensor(name = "fake", value = false)
39        net.chgSensorValue(name = "fake", ids = pos.take(b),
40                           value = true)
41        net.executeMany(node = DemoSecurity, size = 100000,
42                        (n,i) => {})
43      }
44    }
45
```

Simulations results

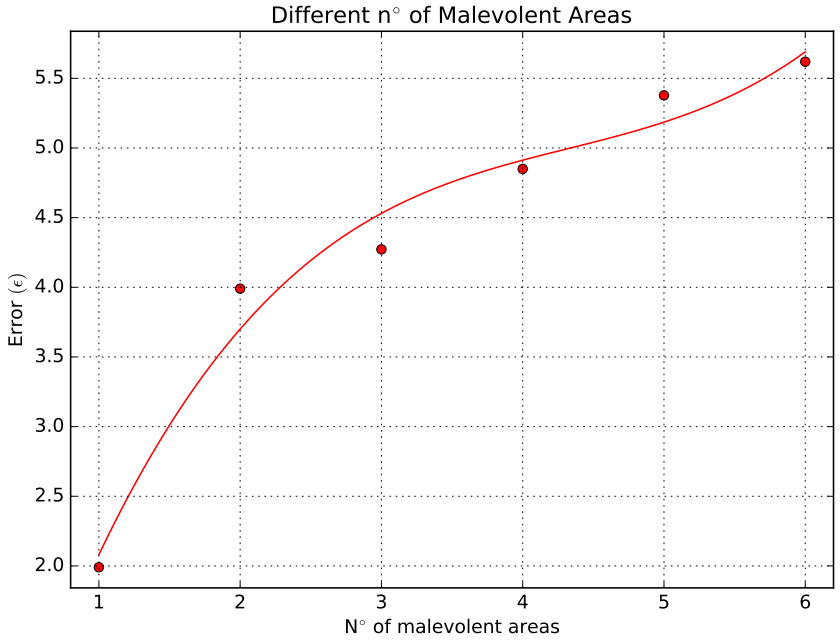

Figure 5.3

As in the experiments done before with a malevolent area of different size,

increasing the amount of fake sources leads to a much broader impact in the error ratio.

5.5.4 Badness

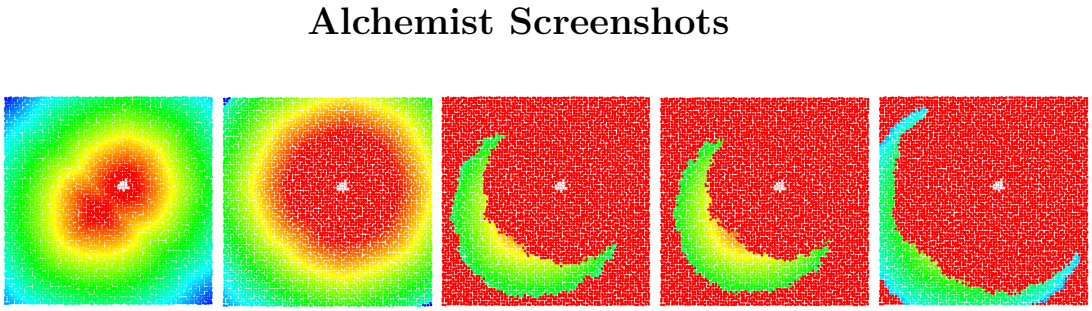

Figure 5.d

From left to right, malevolent nodes send the following corrupted negative values that are $\Big(-1, -5, -15, -25, -100 \Big)$.

The first and the second snapshots are captured after the gradient was stabilized while the following, during *infection* propagation instead. All nodes, in a δ-time proportional to the negative value, will be red. Obviously a negative value has a deleterious effect on a `distance-to` algorithm. However, negative values are easily detectable. An algorithm like `distance-to` where distance between a point A and a point B can't be negative, a simple filter that drops those values solves the problem.

```
1
2  object DemoSecurity extends AggregateProgram with MyLib {
3
4    def isSource = sense[Boolean]("source")
5    def isFake = sense[Boolean]("fake")
6
7    def badness():Double = sense[Double]("badness")
8    def gradient(source: Boolean)(expr:Double): Double = {
9      rep(Double.MaxValue) {
10       distance =>
```

```
11          mux(isFake) { expr }
12          { mux(isSource) { 0.0 } {
13              minHood {
14                nbr { distance } + nbrRange()
15              }
16            }
17          }
18      }
19    }
20
21    def main = gradient(isSource, badness())
22  }
23
24  object DemoSequenceLauncher extends App {
25    def Badness() {
26      val boss = 25*5+15
27      val badness = Array(1.0,5.0,15.0,25.0,100.0)
28
29      for(b <- badness){
30        val net = simulatorFactory.gridLike(n = 15, m = 15,
31                                            stepx = 1, stepy = 1,
32                                            eps = 0.0, rng = 1.1)
33        net.addSensor(name = "source", value = false)
34        net.chgSensorValue(name = "source", ids = Set(25*3+5),
35                           value = true)
36        net.addSensor(name = "fake", value = false)
37        net.chgSensorValue(name = "fake", ids = Set(boss),
38                           value = true)
39        net.addSensor(name = "badness", value = -b)
40
41        net.executeMany(node = DemoSecurity, size = 100000,
42                        (n,i) => {})
43    }
44  }
```

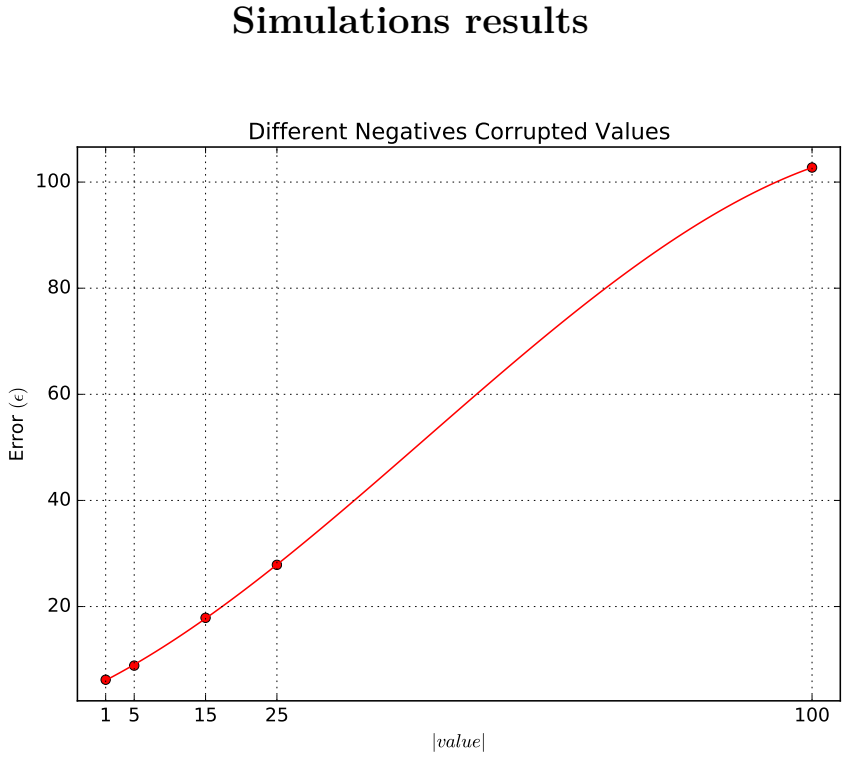

Figure 5.4

Independently from the amount of the negative value, it could be both -10 or -100, this overwhelms other nodes distance value. In the Figure 5.4, the error is proportional to the negative value due to the chose error function.

5.6 Proposed solutions

It is necessary to prevent errors so that each node is aware of data received. Data can be malformed, unstructured or specifically crafted with the purpose of crushing the system or worse, with harmful intentions. An evaluation strategy is needed in order to judge these messages.

Data evaluation can be done in different ways by incorporating an internal trust system or with other precautions. Since aggregate programming scenarios evolve in time, trust values need to be updated accordingly.

Solutions proposed are:

1. The use an **external trust system** that monitors nodes behaviour and that instructs each of them about peers that can be trusted or not.
2. The use of an internal **filter function** based on specific security metrics. For example, in a `distance-to` algorithm it is possible to drop negatives values and/or values that differ too much from other neighbours. Using a threshold η could be useful.
3. The use of a **local trust system**. Each node memorizes a list of untrusted peers that, that is updated accordingly to their behaviour. Security policy and learning algorithms can be used to update the untrusteds' list and to adapt it to different scenarios.

5.6.1 Solution one: External Trust System

From an engineering perspective, by using an external trust system a developer does not need to know how it was developed. Implementation details are decoupled as well as interfaces in OOP. Nodes receive trust fields from the outside, without having to bother to evaluate and update such scores.

Values that come from untrusted nodes will be ignored/discarded.

In order to simulate this solution scenario, it is required to set `Double.PositiveInfinity` to fake (malevolent) peers. It is important to remember that this particular gradient is a `distance-to` function, so distance is equal to the lower value from a neighbour.

`Double.PositiveInfinity` is the least significant value possible and correspond to the furthest hypothetical value.

```
1
2  def gradient(source: Boolean)(expr:Double): Double = {
3    rep(Double.MaxValue) {
4      distance =>
5        mux(isFake) { expr }
6        { mux(isSource) { 0.0 } {
7            minHood { nbr { distance } + nbrRange() }
8          }
9        }
10   }
11 }
12
13 def distanceToWithOracle(untrusted: Boolean): Double = {
14   gradient(isSource) { Double.PositiveInfinity }
15 }
16
```

Simulations Results

Improvements are notable in each of the previous experiments. In the following plots red dots represent error values obtained without the external trust system and the green dots error values with it. Red and green curves are both interpolations.

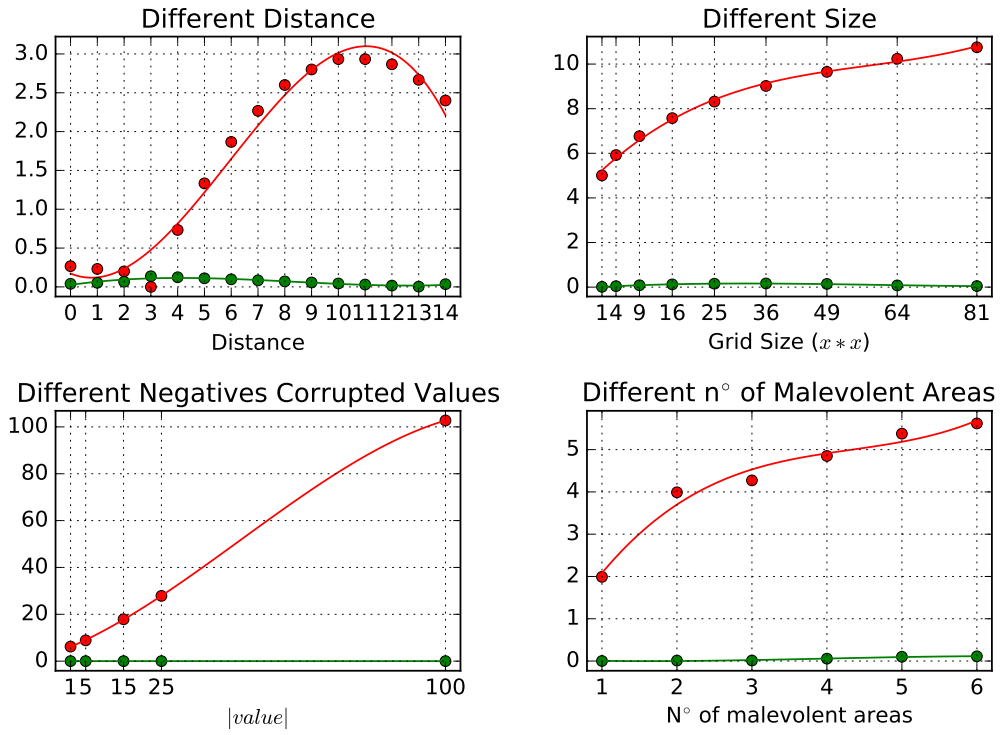

Figure 5.5

5.6.2 Proposed Solution two: Filter Functions

Simulations Results

Remarkable results are feasible with few lines of code using a filter function, builtin in every languages. This techniques alone ward off several security threads. Even a single line of code can set error values to zero in the simulation saw in section 5.5.4.

```
1
2 val nvs = field(nbr(mainb)).filter(_ < 0)
```

This solution preempt malevolent behaviour only by a developer deliberately control. It is improbable to leave aside all possible threads but it deserve attention, at least for the trade-off between spent developing time and results.

5.6.3 Proposed Solution three: Local Trust System

Simulations Results

It is essential to consider that aggregate programming algorithms are reactive to external changes and evolve in time. They dynamically update and each node could move its position in space and change his behaviour. For this reason, a local trust system must be adaptable to changes and be able to recognize erroneous values (i.e. spikes) from good peers and to not increase their trust score.

```
1
2  def distanceToWithLocalTrust()(expr: Double): Double = {
3
4    val threshold = 3  // varies between scenarios, can be trained
5
6    var nvs = getNeighboursField()
7    nvs = nvs.filterUntrusted()
8
9    // Evaluate mean value
10   var mn = mean(nvs.map(r => r._2))
11
12   // Filter new untrusted devices
13   val trusted = nvs.filter(isOutsideThreshold(_._2, threshold, mn))
14
15   // Evaluate new mean value
16   mn = mean(trusted)
17
18   rep( Double.MaxValue ){ _ =>
19     mux(isFake) { expr } { mux(isSource) { 0.0 } { mean } }
20   }
21 }
```

Chapter 6

Conclusion

> "If we wait until we're ready, we'll be waiting for the rest of our lives." - Lemony Snicket, The Ersatz Elevator

Despite this thesis focuses only on a gradient algorithm, it is possible to use the solutions aforementioned to deal with additional security thread in a broader range of aggregate programming algorithms (Section 5.6.1, Section 5.6.2 and Section 5.6.3).

Following the path of these examples and using current technologies discussed above, it is possible to discuss, design and develop new security-aware aggregate programming algorithms.

A noteworthy idea come up during a conversation and it is reported as a roadmap in the following section.

6.1 Future work: Roadmap

Further trust system implementations could exploit the function `recentTrue`:

```
1
2   def recentTrue(state: Boolean, memoryTime: Double): Boolean = {
3     rtSub(timer(10) == 0, state, memoryTime)
4   }
5
```

This function building block is the time-decay aggregate API, **T**, that sums up information throughout time by decaying accumulated values.

```
1
2   def timer[V](length: V)
3               (implicit ev: Numeric[V]) =
4     T[V](length)
5
6   def limitedMemory[V,T](value: V, expValue: V, timeout: T)
7                         (implicit ev: Numeric[T]) = {
8     val t = timer[T](timeout)
9     (if(ev.gt(t, ev.zero)) value else expValue, t)
10  }
11
12  def rtSub(start: Boolean, state: Boolean, memT: Double): Boolean = {
13    if(state) { true }
14    else {
15      limitedMemory[Boolean,Double](start, false, memT)._1
16    }
17  }
```

`recentTrue` let a developer know if a given device is `True` for at least δT time unit. Mixing with proposed solutions (Section 5.6.1, Section 5.6.2, Section 5.6.3), a new version could take into account if a given device behaves badly for a given δT time. If so, the device will be identified as malevolent and banned from the system for a variable period of time.

Appendix 1: Dissertation writing using Pandoc

This dissertation is almost written in Markdown[1]. The formal LaTeX[2] representation document was auto-genereted by Pandoc[3].

6.2 About Pandoc

Pandoc is a universal markup document converter entirely written in Haskell[4]. Pandoc can convert documents in markdown, reStructuredText, textile, HTML, DocBook, LaTeX, MediaWiki markup, TWiki markup, OPML, Emacs Org-Mode, Txt2Tags, Microsoft Word docx, LibreOffice ODT, EPUB, or Haddock markup to

- HTML formats: XHTML, HTML5, and HTML slide shows using Slidy, reveal.js, Slideous, S5, or DZSlides.
- Word processor formats: Microsoft Word docx, OpenOffice/LibreOffice ODT, OpenDocument XML
- Ebooks: EPUB version 2 or 3, FictionBook2
- Documentation formats: DocBook, TEI Simple, GNU TexInfo, Groff man pages, Haddock markup

[1]https://daringfireball.net/projects/markdown
[2]https://www.latex-project.org
[3]http://pandoc.org
[4]https://www.haskell.org

- Page layout formats: InDesign ICML
- Outline formats: OPML
- TeX formats: LaTeX, ConTeXt, LaTeX Beamer slides
- PDF via LaTeX
- Lightweight markup formats: Markdown (including CommonMark), reStructuredText, AsciiDoc, MediaWiki markup, DokuWiki markup, Emacs Org-Mode, Textile Custom formats: custom writers can be written in lua.

Pandoc understands a number of useful markdown syntax extensions, including document metadata (title, author, date); footnotes; tables; definition lists; superscript and subscript; strikeout; enhanced ordered lists (start number and numbering style are significant); running example lists; delimited code blocks with syntax highlighting; smart quotes, dashes, and ellipses; markdown inside HTML blocks; and inline LaTeX. If strict markdown compatibility is desired, all of these extensions can be turned off.

LaTeX math (and even macros) can be used in markdown documents. Several different methods of rendering math in HTML are provided, including MathJax and translation to MathML. LaTeX math is rendered in docx using native Word equation objects.

Pandoc includes a powerful system for automatic citations and bibliographies, Many forms of bibliography database can be used, including bibtex, RIS, EndNote, ISI, MEDLINE, MODS, and JSON citeproc. Citations work in every output format.

Pandoc includes a Haskell library and a standalone command-line program. The library includes separate modules for each input and output format, so adding a new input or output format just requires adding a new module.

Pandoc is free software, released under the GPL by John MacFarlane.

6.3 Motivations

Science uses writing since the early days to pass on knowledge to the new generations. Internet and the World Wide Web have drastically aids the way people communicate and share information.

English is adopted as a lingua franca in order to reach more readers.

It is essential, however, to reach also other types of *readers*, blind people. Technologies such as screen readers help them to access digital information but some data presentation formats are more accessible than others. The widely used PDF format, for example, is really tricky to read. This is the real motivation behind the use of a more accessible format such as Markdown or HTML: being accessible as much as possible.

This is not a call to dismiss working with LaTeX and related generated PDF documents. They have a really well structured format, awesome for printing. This is a call to all academics to try to be aware as much as possible of alternative solutions that help them in their vocation: spread their knowledge to the world.

References

Abelson, H. et al., 2000. Amorphous computing. *Commun. ACM*, 43(5), pp.74–82. Available at: http://doi.acm.org/10.1145/332833.332842.

akka.io, [Online; accessed: March 2017]. Akka toolkit.

Ashby, W.R., 1947. Principles of the self-organizing dynamic system. *The Journal of general psychology*, 37(2), pp.125–128.

Bar-Yam, Y., 1997. *Dynamics of complex systems*, Addison-Wesley Reading, MA.

Beal, J. & Bachrach, J., 2006. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2), pp.10–19. Available at: http://dx.doi.org/10.1109/MIS.2006.29.

Beal, J. & Viroli, M., 2016. Aggregate programming: From foundations to applications. In M. Bernardo, R. D. Nicola, & J. Hillston, eds. *Formal methods for the quantitative evaluation of collective adaptive systems - 16th international school on formal methods for the design of computer, communication, and software systems, SFM 2016, bertinoro, italy, june 20-24, 2016, advanced lectures*. Lecture notes in computer science. Springer, pp. 233–260. Available at: http://dx.doi.org/10.1007/978-3-319-34096-8_8.

Beal, J. & Viroli, M., 2014. Building blocks for aggregate programming of self-organising applications. In *Self-adaptive and self-organizing systems workshops (sasow), 2014 ieee eighth international conference on*. IEEE, pp. 8–13.

Beal, J., Pianini, D. & Viroli, M., 2015. Aggregate programming for the internet of things. *IEEE Computer*, 48(9), pp.22–30. Available at: http://dx.doi.org/10.1109/MC.2015.261.

Britannica, E., Field (physics). Available at: https://www.britannica.com/science/field-physics.

Cho, J.-H., Swami, A. & Chen, R., 2011. A survey on trust management for mobile ad hoc networks. *IEEE Communications Surveys & Tutorials*, 13(4), pp.562–583.

Clark, S.S., Beal, J. & Pal, P., 2015. Distributed recovery for enterprise services. In *Self-adaptive and self-organizing systems (saso), 2015 ieee 9th international conference on*. IEEE, pp. 111–120.

Damiani, F., Viroli, M. & Beal, J., 2016. A type-sound calculus of computational fields. *Sci. Comput. Program.*, 117(C), pp.17–44. Available at: http://dx.doi.org.ezproxy.unibo.it/10.1016/j.scico.2015.11.005.

Dictionary, C., [Online; accessed: March 2017]. Trust. Available at: http://dictionary.cambridge.org/dictionary/english/trust?a=british.

Foerster von H, C., 1987. Encyclopedia of artificial intelligence. *NY: John Wiley and*

Sons, 19 7.

Francia, M., [Online; accessed: March 2017]. Protelis-sandbox (w4bo).

github.com/jak3/Protelis-Sandbox, [Online; accessed: March 2017]. Protelis-sandbox (j4ke).

gradle.org, [Online; accessed: March 2017]. Gradle homepage.

Gray, J., 1986. Why do computers stop and what can be done about it. In *Symposium on reliability in distributed software and database systems*. Los Angeles, CA, USA, pp. 3–12.

Haken, H., 1977. Synergetics. *Physics Bulletin*, 28(9), p.412.

Hewitt, C. & Jong, P. de, 1984. Open systems. In M. L. Brodie, J. Mylopoulos, & J. W. Schmidt, eds. *On conceptual modelling: Perspectives from artificial intelligence, databases, and programming languages*. New York, NY: Springer New York, pp. 147–164. Available at: http://dx.doi.org/10.1007/978-1-4612-5196-5_6.

Heylighen, F. & others, 2001. The science of self-organization and adaptivity. *The encyclopedia of life support systems*, 5(3), pp.253–280.

Jøsang, A., 2007. Trust and reputation systems. In *Foundations of security analysis and design iv*. Springer, pp. 209–245.

Jøsang, A. & Presti, S.L., 2004. Analysing the relationship between risk and trust. In *International conference on trust management*. Springer, pp. 135–145.

Kamvar, S.D., Schlosser, M.T. & Garcia-Molina, H., 2003. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th international conference on world wide web*. ACM, pp. 640–651.

Lamport, L., 1977. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2), pp.125–143.

Mármol, F.G. & Pérez, G.M., 2009. Security threats scenarios in trust and reputation models for distributed systems. *computers & security*, 28(7), pp.545–556.

McMullin, E., 2002. The origins of the field concept in physics. *Physics in Perspective*, 4(1), pp.13–39. Available at: http://dx.doi.org/10.1007/s00016-002-8357-5.

Nicolis, G., Prigogine, I. & others, 1977. *Self-organization in nonequilibrium systems*, Wiley, New York.

Paulos, A. et al., 2013. Isolation of malicious external inputs in a security focused adaptive execution environment. In *Availability, reliability and security (ares), 2013 eighth international conference on*. IEEE, pp. 82–91.

Pianini, D., 2017. Github.com/protelis/protelis. Available at: https://github.com/Protelis/Protelis.

protelis.github.io, [Online; accessed: March 2017]. Protelis language homepage.

Solhaug, B., Elgesem, D. & Stølen, K., 2007. Why trust is not proportional to risk. In *Proceedings of the 2nd international conference on availability, reliability and security (ares'07)*. pp. 11–18.

Tanenbaum, A.S. & Van Steen, M., 2007. *Distributed systems*, Prentice-Hall.

Tokoro, M., 1990. Computational field model: Toward a new computing model/methodology for open distributed environment. In *Distributed computing*

systems, 1990. proceedings., second ieee workshop on future trends of. pp. 501–506.

www.groovy-lang.org, [Online; accessed: March 2017]. Groovy homepage.

Xiong, L. & Liu, L., 2004. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE transactions on Knowledge and Data Engineering*, 16(7), pp.843–857.

Zambonelli, F. et al., 2011. Self-aware pervasive service ecosystems. *Procedia Computer Science*, 7, pp.197–199.