Alma Mater Studiorum · Università di Bologna
Campus di Cesena

Scuola di Ingegneria e Architettura

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

A Reinforcement Learning approach to discriminate unsafe devices in aggregate computing systems

Relatore:
Prof. Mirko Viroli

Corelatore:
Dott. Roberto Casadei

Presentata da:
Chiara Volonnino

Sessione III
Anno Accademico 2018/1019

"Knowing is not enough; we must apply.
Willing is not enough; we must do."
Cit. Leonardo Da Vinci

Contents

List of Figures

List of Code

Summary of Notation

S	Set of states
$A(s_t)$	Set of actions that can be taken in state s
R	Reward value, it can be positive, negative or zero
s	Actual agent state
s'	Next agent state
a	Actual action selected by an agent
r	Reward value
t	Discrete time step
T	Final time step
π, π_*	Policy and optimal policy
$v_\pi(s)$, $v_*(s)$	Value function and optimal value function
$q_\pi(s,a)$, $q_*(s,a)$	State-value function and optimal state-value function (Q-function)
γ	Discounted rate
E	Expected return value
$p(x)$	Probability density function when x is continuous
$P(x)$	Probability mass function when x is discrete
$P(x\|y)$	Conditional probability of x given y
$\xrightarrow{E}$	Complete policy evaluation
$\xrightarrow{I}$	Complete policy improvement
α	Learning rate

Abstract

Reinforcement learning is a machine learning approach that has been studied for many years, but particularly nowadays the interest about this topic has exponentially grown. Its purpose is to create autonomous agents able to sense and act in their environment. They should learn to choose optimal actions to achieve their goals, in order to maximise a cumulative reward. Based on the knowledge of the environment, there are different solutions.

Aggregate programming is a paradigm that supports the large-scale programming of adaptive systems by focusing on the behaviour of the cluster instead of the singles. One promising aggregate programming approach is based on the field calculus, that allows the definition of aggregate programs by the functional composition of computational fields. A topic of interest related to Aggregate Computing is computer security. Aggregate Computing systems are, in fact, vulnerable to security threats due to their distributed nature, situatedness and openness, which can make participant nodes leave and join the computation at any time.

A solution that enables to combine reinforcement learning, aggregate computing and security, would be an interesting and innovative approach, especially because there are no experiments so far that include this combination.

The goal of this thesis is to implement a Scala library for reinforcement learning, which must be easily integrated with the aggregate computing context. Starting from an existing work, on trust computation in aggregate applications, we want to train a network, via reinforcement learning, which

through the calculation of the gradient – a fundamental pattern of collective coordination – is able to identify and discriminate compromised nodes.

The dissertation work focused on: (*i*) development of a generic Scala library that implements the reinforcement approach, in accord to an aggregate computing model; (*ii*) development of a reinforcement learning based solution; (*iii*) integration of the solution that allows us to calculate the trust gradient.

`Keywords`: Aggregate Programming, Reinforcement Learning, Monte Carlo Learning, Distributed Learning, Scala, Scafi, Alchemist.

Introduction

Reinforcement learning is a machine learning approach that has been studied for many years, but particularly nowadays the interest about this topic is exponentially growing. Its purpose is to create autonomous agents able to sense and act in their environment. They should learn to choose optimal actions to achieve their goals, in order to maximise a cumulative reward. Based on the knowledge of the environment, there are different solutions. Among the most known and used algorithm, we remember: Q-learning and Monte Carlo learning.

Aggregate programming is a paradigm that supports the large-scale programming of adaptive systems, by focusing on the behaviour of the cluster rather than of the single ones. One promising aggregate programming approach is based on the field calculus, that allows the definition of aggregate programs by the functional composition of computational fields. A topic of interest related to Aggregate Computing is computer security. Aggregate Computing systems are, in fact, vulnerable to security threats due to their distributed nature, situatedness and openness, which can make participant nodes leave and join the computation at any time.

A solution that enables to combine reinforcement learning, aggregate computing and security is an interesting and innovative approach, especially because there are no experiments so far that include this combination.

The goal of this thesis is to implement a Scala library for reinforcement learning. This library must be easily integrated with the aggregate computing context. Starting from an existing work, we want to train a network,

via reinforcement learning, which, through the calculation of the gradient – a fundamental pattern of collective coordination – is able to identify and discriminate compromised nodes.

The dissertation work focused on:

1. the development of a generic Scala library that implements the reinforcement approach, in accord to an aggregate computing model;

2. the development of a reinforcement learning based solution;

3. the integration of the solution that allows us to calculate the trust gradient.

It is notable that the initial design of the library has been authored by the supervisors.

This thesis is conceptually organised into three parts.

In the first part, a background overview is provided. In chapter 1, *Reinforcement Learning*, the reinforcement learning approaches are introduced, focusing on: main elements, different existing models and most famous algorithms, like Q-learning and Monte Carlo learning. Then, the multi-agent reinforcement learning is discussed. In chapter 2, *Aggregate Computing*, a high-level description of the aggregate programming is offered. An outline of the computational field is supplied and the aggregate programming stack, based on the field calculation, is described.

The second part, which represents the main work of this thesis, covers the implementation of the reinforcement learning library and its application in an aggregate computing context in order to discriminate unsafe devices. In chapter 3, *Analysis*, the requirements and the problem are cleared up. Then, in chapter 4, *Design*, the architecture and the key elements of the design of our prototype are outlined, while a more detailed view of the solution is provided in chapter 5, *Implementation*.

Finally, the evaluation of the work is carried out in the last part. In chapter 6, *Evaluation*, an assessment of our prototype compared to the re-

quirements and some experiments are reported. Then, included in chapter 7, *Conclusion*, some general, retrospective considerations.

Chapter 1

Reinforcement Learning

In this chapter we will make an overview of the reinforcement learning background, based on [20] and [3]. At first we will focus on the fundamentals, like policy, function value and optimal condition, in a more specific way for Markov decision processes. Then, we will cover some of the most famous algorithms such as Dynamic Programming, Monte Carlo learning and Q-learning. In the end we will consider reinforcement learning in a multi-agent environment.

1.1 Learning Techniques

We will define *learning programs* as all computer programs which improve their performances at some tasks through their experiences. More formally: a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [18].

In a more specific way we can talk about three different learning techniques:

- *Supervised Learning*: a set of labeled examples representing input-output relationships, is used to derive a model for discovering unknown relationships, is provided by a knowledgeable external supervisor.

- *Unsupervised Learning*: the system is provided with a series of inputs, representing the experience of the system, which must classify and organise in order to find structures hidden in collections of unlabeled data.

- *Reinforcement Learning*: an agent able to sense and act in its environment should learn to choose optimal actions to achieve its goal.

Reinforcement Learning (RL) is a type of associative learning where a relationship between stimulus and response is studied. In classical conditioning, instead, is considered the relationship between two stimulus. Essentially this type of learning was born to learn from interactions.

1.2 Elements of Reinforcement Learning

According to [3] and [20], in reinforcement learning there is a decision maker called **agent**, which interacts with its **environment**. Notably, as shown in fig. 1.1, at time $t = 0, 1, 2, \ldots$ the agent is in **state** $s_t \in \mathcal{S}$ and chooses the **action** $a_t \in \mathcal{A}(s_t)$. Then the agent receives a signal, $r_{t+1} \in \mathcal{R}$, called **reward**, and moves to the next state s_{t+1}. The reward can be positive or negative. A positive reward is applied for increasing the action probability to be chosen, on the contrary the action probability is decreased by negative reward. This

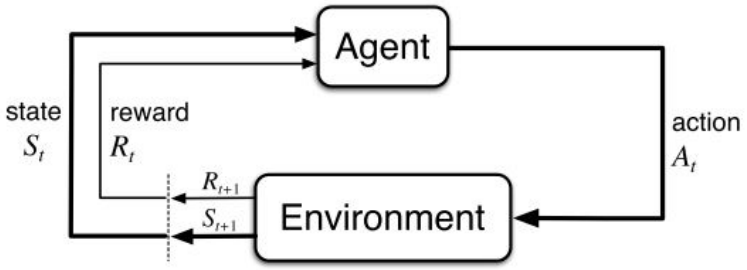


Figure 1.1: The agent–environment interaction in a Markov decision process [20].

problem is modeled using the *Markov Decision Problem* (MDP), discussed

in section 1.3. Actually in reinforcement learning a solution is a sequence of actions, and the agent should learn the best sequence in order to maximise its total reward. The sequence of actions taken, from the first to the last state (usually called *terminal state*), embodies an *episode* (or trial).

The **policy** $\pi : \mathcal{S} \to \mathcal{A}$ delineates the mapping from the perceived states of the environment to the actions that can be taken, in other words the policy defines the agent's behaviour. While reward represents the signal received in the immediate sense, the **value function** $v_\pi(s)$ specifies what is good in the *long run*. In this context, the agent's objective is to learn a policy with a view to maximise the expected return value. For the *finite-horizon model* this value is defined by:

$$v_\pi(s_t) = E_\pi[R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T] = E_\pi\left[\sum_{k=0}^{T} R_{t+k}\right] \tag{1.1}$$

However, the eq. (1.1) is problematic for continuous tasks, also there is no prefixed limit to the episode. To solve this issue we use the *infinite-horizon discounted model*:

$$v_\pi(s_t) = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-1} R_T] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] \tag{1.2}$$

where γ is the *discount rate* and $0 \leq \gamma \leq 1$.

Similarly, we call **state-value function** (or **Q-function**) the expected return, which is obtained starting from state s, taking the action a by following the policy π. It is denoted $q_\pi(s, a)$ and is defined by:

$$q_\pi(s, a) = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-1} R_T] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] \tag{1.3}$$

1.2.1 Optimal Policy

A policy π is considered better than a policy π' if its expected return is better than π' expected return for each $s \in \mathcal{S}$, which means that $v_\pi(s) \geq v_{\pi'(s)}$ for each $s \in \mathcal{S}$. If π is better than π', we can assert that π is an *optimal policy*, and we denote it as $\pi*$ [20]. There might be more than one optimal policy and

they necessary share the same state-value function, which is called *optimal state-value function*, denoted v_* and defined as:

$$v_*(s) = \max_\pi v_\pi(s) \qquad \forall s \in \mathcal{S} \tag{1.4}$$

Optimal policies also share the same *optimal action-value function* q_*, defined as:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \qquad \forall s \in \mathcal{S}, \forall s \in \mathcal{A}(s) \tag{1.5}$$

It follows that for an optimal policy:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_*(s,a) \qquad \forall s \in \mathcal{S}, \forall s \in \mathcal{A}(s) \tag{1.6}$$

1.2.2 Exploration vs. Exploitation

One critical point of reinforcement learning is the *trade-off* between exploration and exploitation. In fact, in order to obtain an high reward, an agent must prefer actions it has already selected in the past and which have produced good results (**exploitation**), but to figure out which these actions are, it must also try actions it has never selected before in order to make better action selections in the future (**exploration**). The best case is when the agent tries a variety of actions and progressively favours those that appear to be the best.

The exploration–exploitation dilemma has been intensively studied by mathematicians for many decades, it still remains unresolved [20].

1.3 Markov Decision Process

Reinforcement learning methods typically represent the environment in the form of the Markov decision process (MDP). The main difference between classical dynamic programming methods and reinforcement learning algorithms is the lattice, which does not assume the same knowledge of an exact mathematical model. In fact it does not matter to know how it is

exactly done but, more generally, we will define an overall assumption about its properties.

A Markov Decision Process is formally defined by [20]:

- $\mathcal{S}$: finished set of states.
- $\mathcal{A}$: finished set of actions.
- $\mathcal{T}$: transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{S})$, which assigns to each action-state pair a probability distribution on S.
- $\mathcal{R}$: reinforcement function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{R}$, which assigns a numerical value at each possible transition.

As shown in fig. 1.1 the interaction between agent and environment occurs in the following way: at time t the agent senses the environment as s_t, based on this value it takes an action a_t and the environment responses with an immediate reward $r_{t+1} = r(s_t, a_t)$. Then it changes its state $s_{t+1} = \delta(s_t, a_t)$ where δ and r are part of the environment and they are not necessarily known by the agent, they can also be non-deterministic.

Markov property

A problem is defined by the Markov property if:

$$P[s_{t+1} = s, r_{t+1} = r | s_t, a_t] = P[s_{t+1}, r_{t+1} | s_t, a_t, r_t, \ldots, r_1, s_0, a_0] \qquad (1.7)$$

which involves that the probability of each possible value of s_{t+1} and r_{t+1} depends only on the predecessors state and action, s_t and A_t [20].

1.4 Model-based Learning

In the *model-based* model we completely know the environment model parameters $p(s_{t+1}, r_{t+1} | s_t, a_t)$ and $P(s_{t+1}, r_{t+1} | s_t, a_t)$. Typically we do not need any exploration and we can immediately solve the optimal policies and optimal value-function by *dynamic programming*.

1.4.1 Dynamic Programming

Dynamic programming is a problem solving technique introduced in the 1940s by the american mathematician *Richard Bellman*. It refers to a collection of algorithms that can be used to compute optimal policies when a perfect model of the environment, like a Markov decision process, is given.

Classical dynamic programming methods are limited used in reinforcement learning, because they are computational expansive and a lot of their assumptions include a perfect environment model. They are, however, very important from a theoretical point of view. As the matter of fact, some of reinforcement learning methods use value-functions to organise and structure the search of good policies, like is done in dynamic programming, by using the *Bellman equation* [20].

Bellman Equation

The Bellman equations formulate the problem of the maximising the sum of the expected reward in terms of its recursive relationship with the value-function. Joining together eq. (1.6) with eq. (1.3) we get:

$$\begin{aligned} v_* &= \max_a E_{\pi_*}[G_t | s_t = s, a_t = a] \\ &= \max_a E_{\pi_*}\left[R_t + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s, a_t = a\right] \\ &= \max_a \sum_{s'} p(s'|s,a)[R_t + \gamma v_*(s')] \end{aligned} \tag{1.8}$$

where G_t is the expected return and $p(s'|s,a)$ is the probability of arriving in state s' starting from state s taking action a. In the first step we break down the G_t value recursively (strength of the equation).

For q_* the optimal equation is defined by eq. (1.3) and eq. (1.5) is:

$$\begin{aligned} q_*(s,a) &= E[R_t + \gamma \max_{a'} q_*(s_{t+1}, a') | s_t = s, a_t = a] \\ &= \sum_{s'} p(s'|s,a)[R_t + \gamma \max_{a'} q_*(s', a')] \end{aligned} \tag{1.9}$$

1.5 Model-free Learning

Opposite to model-based learning there is *model-free* learning, in which the prediction estimates the value function of an unknown MDP.

The three main methods for model-free predictions are:

- Monte-Carlo Learning
- Temporal-Difference Learning
- TD(λ)

1.5.1 Monte Carlo Learning

The Monte Carlo methods are used to solve reinforcement learning problems, it employs averaging sample returns for episodic tasks.

The idea is to split the experience in episodes, each episode can eventually terminate no matter what actions are selected [20]. Only when an episode terminates the policy and the value -unction change. Summarising, at first the state-value function from the experience is estimated, then is computed the average of the returns obtained after visiting a certain state. As many returns are observed as more the average should converge to the expected value.

First-visit vs Every-visit

We defined as *visit* to s each occurrence of state s in an episode. We have two approaches to calculate the expected reward, depending on how it is estimated $v_\pi(s)$ [similarly $q_\pi(s, a)$].

- **First-visit** (FV): estimates $v_\pi(s)$ as the average of the returns for the first visit to s.
- **Every-visit** (EV): estimates $v_\pi(s)$ as the average of the returns for all the visits to s.

Example:
Two episodes are given:

1. A + 1 → A + 3 → B + 4 → A − 1 → B − 1 → end
2. B − 1 → A + 3 → B − 1 → A − 1 → end

where $s+r \to s'$ represents a transition from state s to state s' with a reward r.

Table 1.1: Calculate $v(s)$ using first-visit approach

FV	Episode 1	Episode 2	v(s)
A	$(+\ 1 + 3 + 4 - 1 - 1) = 6$	$(+\ 3 - 1 - 1) = 1$	$\frac{1}{2}(6+1) = 3,5$
B	(+ 4 - 1 - 1) = 2	(- 1 + 3 - 1 - 1) = 0	$\frac{1}{2}(2+0) = 1$

Table 1.2: Calculate $v(s)$ using every-visit approach

EV	Episode 1	Episode 2	v(s)
A	$(+\ 1 + 3 + 4 - 1 - 1) = 6$ $(+\ 3 + 4 - 1 - 1) = 5$ $(-\ 1 - 1) = -\ 2$ total: 9	$(+\ 3 - 1 - 1) = 1$ $-\ 1$ total: 0	$\frac{1}{5}(9+0) = 1,8$
B	$(+\ 4 - 1 - 1) = 2$ $-\ 1$ total:$-\ 1$	$(-\ 1 + 3 - 1 - 1) = 0$ $(-\ 1 - 1) = -\ 2$ total: $-\ 2$	$\frac{1}{4}(-1-2) = -0,25$

Exploring Start

When a model of the environment is present the estimate of the action-value function is obtained by simply computing a policy like in dynamic programming. Vice versa, if the model is not defined, the state-value alone is not sufficient because some (s, a) pairs may never be visited. To solved this problem Monte Carlo learning adopts the **exploring start** (ES), which

specifies that the episodes started in (s, a) pair and that every pairs have a non-zero probability of being selected as the start.

The pseudo-code of the Monte Carlo exploring start algorithm with first-visit approach is available in fig. 1.3.

This approach sometimes is useful but generally it can not be relied, particularly when the agent learns directly from actual interactions with the environment. In this case stochastic policies, which have non-zero probability of selecting all actions in each state, are considered the best option. [20].

Generalised Policy Iteration

Generalised Policy Iteration (GPI) is an iterative schema based on two processes (fig. 1.2):

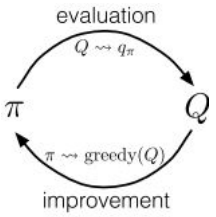


Figure 1.2: Monte Carlo control [20].

- **Policy evaluation**: builds an approximation of the value-function relying on the current policy.

- **Policy improvement**: improves the current policy relying on the current value-function. Particularly the improvement is done by making a *greedy* policy respect the current value-function.

It is proven that, begin with an arbitrary policy π_0 will end with an optimal policy π_* and an optimal action-value function q_*: $\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \ldots \xrightarrow{I} \pi_* \xrightarrow{E} q_*$.

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Figure 1.3: Monte Carlo exploring start using first visit pseudo-code [20].

1.5.2 Temporal-Difference Learning

Temporal-difference (TD) methods are a combination of Monte Carlo and dynamic programming ideas. After all, temporal-difference techniques can learn directly from experience; they evaluate the policy by calculating the *temporal error*, which is the difference between the new estimate and the old estimate of the value-function. The value-function updating equations is:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{1.10}$$

where α is the **learning rate**. According to [20], when the update target is $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ we are in the case of **TD(0)**, or *one-step* TD, methods, a special case of TD(λ) where $\lambda > 1$. Since TD(0) bases its update only on a part of the existing estimates, we say that it is a *bootstrapping* method, like DP. Therefore temporal-difference algorithms approximate the value-function at each time step t, in which a non-terminal state is visited, but the value-function is changed only once, by the sum of all increments. This type of update is called *batch updating*: it reduces the variance but increases the bias in the estimate of the value-function.

Q-learning

One of the most famous temporal-difference algorithms is Q-learning, which is an off is an *off-policy* algorithm because it directly approximates Q regardless of the policy it is following.

Starting from the eq. (1.10) we obtain for the Q-learning method:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (1.11)$$

The algorithm is described in fig. 1.4.

Usually a table is used in order to store the value-function values, but the more the number of the states and actions increases the more this algorithm is computationally inefficient. To solve this problem function approximators have been introduced.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 1.4: Q-learning pseudo-code [20].

1.6 Multi-agent Reinforcement Learning

The problem of reinforcement learning in **multi-agent systems** (MAS) has been studied for a vary long time, most widely over recent years, carried out with a lot of experiments. All these studies figured out that the collaboration of several agents can bring a benefit to the agents community, improving the

performances, the accuracy and convergence of the problem solution [20]. Furthermore, MAS has connections to social and collective aspects of the agent behaviour. Obviously, those systems are usually formed by a large number of agents whose involve an high computation time and the need of keeping a large amount of data memorised; without forgetting that we are dealing, by definition, with strongly decentralized and partially observable architectures.

In multi-agent systems all the agents try to maximise a common reward they simultaneously receive. This scenario is called *cooperative game* (or *team problem*). This means that each individual agent has only limited ability to affect the reward, because each single agent contributes just for one component of the collective action; therefore an agent behaviour depends on how all the other agents are behaving [20].

For those reasons, two main troubles have arisen: how to decrease the total number of episodes necessary to make the agents learn how to solve the problem and how to decrease the number of stored data for each agent.

1.6.1 Related Works

Some works that have tried to solve, in different ways, the above mentioned problems.

Architectural Solutions

Some solutions, to improve accuracy and performances, include two types of agents, according to the *master-slave architecture*, [7], [14], [16], [23]. Has been proven that the convergence of these solutions is better and more efficient than solutions that do not include the use of master-slave architecture.

It is also clear, as shown in [11], [16], [10], [17], [9], [23], that an agent's decision is conditioned by its neighbours' behaviours. Others, propose to use *message passing* to increase self-organization and coordination between agents, like in [11], in which they have done so using the joint-value function approximation as a linear combination of local-value functions. This function

of factored values allows the agents to globally calculate the optimal joint-action using a very natural message passing scheme.

Conceptual Solutions

In [12], to reduce the amount of stored data, they propose to use a state aggregation, that, according to some constraints of similarity and closeness between different states, allows coupling and processing the values of those states like it were only one; so you will always have the same return value for every processed action in that subset of states. A similar solution is introduced in [23] which poses the problem as a *clustering problem*, in this case we want to pair states and improve communication between different agents.

Some solutions, [15], [11], [10], [9], in order to reduce the number of data, suggest to simply keep the best value or the most probable in the value function and discarding the others.

Challenges

However, it remains clear that the question of which is the best approach is still open. A recent challenge, called *Challenges and Opportunities for Multi-Agent Reinforcement Learning* (COMARL) [8], has been issued by AAAI Spring Symposium Series in cooperation with the Stanford University Computer Science Department. COMARL's goal is to bring together researchers from all over the world to try to solve many of the challenges that are still open in multi-agent reinforcement learning.

Chapter 2

Aggregate Computing

In this chapter we will introduce the main issues concerning aggregate computing an why has been necessary to move towards this approach. Then, the ScaFi framework and the Alchemist simulator will be presented and analysed. At last we overview the relation between Aggregate Computing and computer security.

2.1 Collective Adaptive Computing

The exponential growth of distributed networked IT devices brought to the world of the Internet of Things (IoT). The need of interaction and cooperation between those devices have been two of the main reasons that made bring together our day life objects with the digital world. To do so the most natural paradigm is *single-device viewpoint*, but even if it is useful, for certain applications it still has some limits; in fact it was proved to be inappropriate due to the increasing number of devices any the strong bond it has with their spatial concept. [13].

To solve those issues there are several possible methods, one of them is **Aggregate Computing** (AC). According to [13], [6], aggregate computing is a *large scale* approach to distributed systems programming with which it is possible to define a *generic collective behaviour* of the system in high-level

and modular way. Furthermore, this approach supports the programming of self-adaptive/self-organising behaviours.

2.2 Aggregate Computational Model

An aggregate system is a compound of *computational devices*. All of these devices perform the same aggregate program in asynchronous rounds of computation and define an *export* that represents its last state value that will be propagate at the neighbours during the executions.

In each round is defined by four steps, as indicated in [6]:

- *Creation of the execution context state* which includes the latest computed local value, the most recent exports received from neighbours, and a snapshot of local sensor values.
- *Local execution of the aggregate program*, which, based on context state, yields the new state (or export).
- *Propagation of the computed export to the entire neighbourhood*, done by a broadcast.
- *Activation of the actuators*, with the input given by the result of computation.

The aggregate computing idea, as shown in fig. 2.1, is to program the aggregate not individual device, and it is based on three main fundamentals, reported in [13]:

- the "machine" being programmed is a region of the computational environment whose specific details are abstracted away-perhaps even to a pure spatial continuum;
- the program is specified as manipulation of data constructs with spatial and temporal extent across that region;

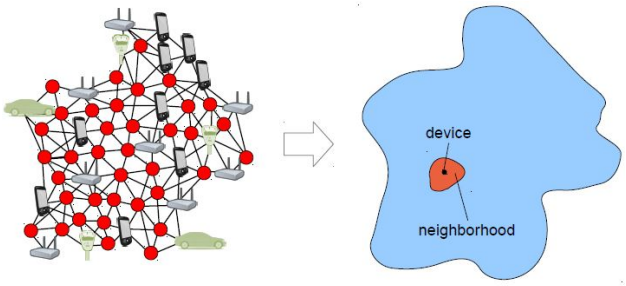


Figure 2.1: Physical sight according to computational fields logic of a network of devices [21].

- these manipulations are actually executed by the individual devices in the region, using resilient coordination mechanisms and proximity based interactions.

In this context, the reference elaboration process can be seen as an atomic manipulation of a collective data structure and the model can be described by the *computational fields* and the *field calculus*. [6]

2.2.1 Computational Field

A **computational field** (fig. 2.1) is a distributed data structure that associates a value to each device localised into the space-time. More formally, a field as a space-time structure is defined $\phi : D \rightarrow V$, where:

- D is the events domain. When an event E is defined $E = \langle \delta, t, p \rangle$ which consists of device δ observed at time t in position p.

- V is the field value and can be any data value.

2.2.2 Aggregate Programming Stack

The aggregate computing stack, as shown in fig. 2.2, is built as a multi-layer structure. Following a brief description is provided.

Device Capabilities

In the low layer the devices capabilities are abstracted and composed with the purpose of creating a common level.

Field Calculus

Field calculus [22] is a theoretical model used to describe a set of primitives to manipulate a computational field in a global and expressive way. In particular we have:

- *Function*: $b(e_1, \ldots, e_n)$.
  Function b is applied to inputs $e_1, \ldots, e_n$.
- *Dynamic*: $rep(x \leftarrow v)\{s_1; \ldots; s_n\}$.
  Defines a local variable x, initialised by the value v and periodically updated with the results obtained by computing $s_1, \ldots, s_n$.
- *Interaction*: $nbr(s)$.
  Defines a map that associates each device in the neighbourhood to its last value s.
- *Restriction*: $if(e)\{s_1; \ldots; s_n\}else\{s'_1; \ldots; s'_n\}$.
  This operator separates the network in different regions. Each region, based on e condition, runs different programs.

These constructs allow portability, infrastructure independency and modularity.

Building Blocks

In the third layer there is the *building block*, it identifies a generic collection in order to add a resilience coordination layer to the infrastructure. Many building blocks can be combined to create advanced applications using aggregate programming of collective systems [13]. There are four types of building blocks:

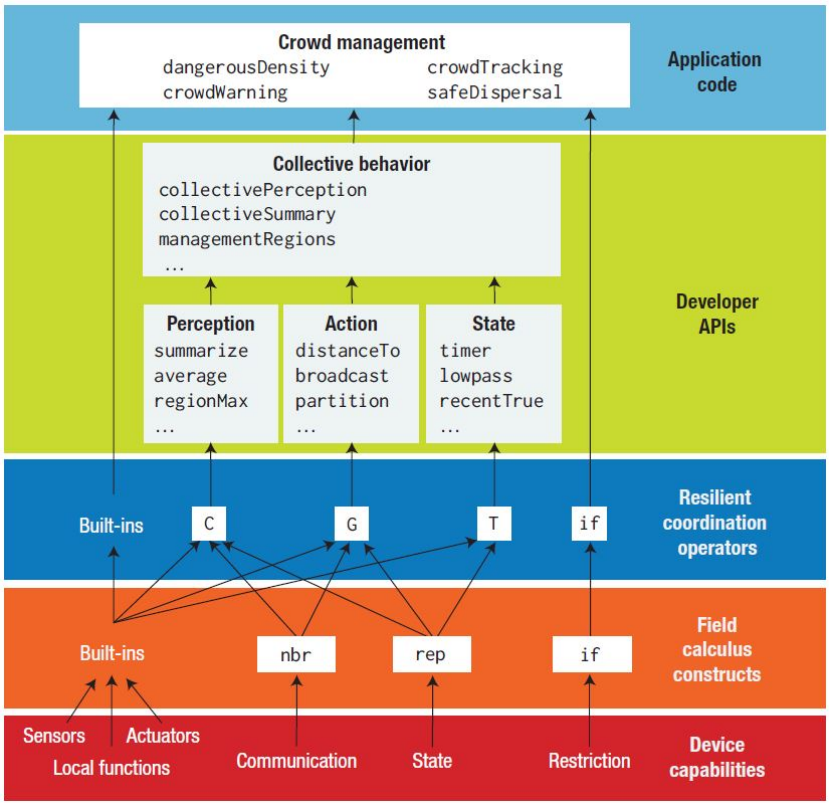


Figure 2.2: Aggregate programming stack [13].

- *Gradient-cast*: `G(source, init, metric, accumulator)`.
  It spreads information across the space through distance estimation and broadcast.

- *Converge-cast*: `C(potential, accumulate, local, null)`.
  It collects information through the space and accumulates values up to a potential field.

- *Time-decay*: `T(initial, decay)`.
  Information across time.

- *Sparse-choice*: `S(grain, metric)`.
  It is useful to create partitions and to select subsets of devices.

All those building blocks are *self-stabilising*, this means that all mecha-

nisms of this level can reach the corrected state in a finite number of steps, regardless of their initial state [4].

Developer APIs

The second-to-last layer provides a *user-friendly API*. These developer APIs are resilient and safely composable since they rely on resilient operators and field calculus constructs. They can, in turn, be combined in order to raise the abstraction level and ease the development of applications for IoT scenarios [13].

2.3 ScaFi

ScaFi[1] (Scala Fields) is a Scala-based library and framework born for the purpose of providing an integrated environment to code aggregate systems, exploiting the advantages of the Scala programming language. ScaFi implements a language for field calculus, embedded within Scala as an internal *domain specific language* (DSL), and provides a platform and APIs to simulate and execute aggregate applications. Moreover, it offers a platform which supports both the definition and the execution of distributed aggregate applications [2].

More details about ScaFi can be found in its official documentation [2].

Why Scala?

Scala is a modern language interoperable with Java, it integrates the object-oriented and functional paradigms in an excellent way and offers advanced features to support the design of high-level software libraries and fluent APIs [6].

[1] https://github.com/scafi/scafi

2.4 Alchemist

Alchemist[2] is a simulator for pervasive, aggregate and nature-inspired computing.

As shown in fig. 2.3, in Alchemist, the simulation world is composed of the following entities [19]:

- *Molecule*: an abstraction representing the node's information.
- *Concentration*: the value associated to a particular molecule.
- *Node*: a container of molecules and reactions which live inside an environment.
- *Environment*: space abstraction which contains the nodes.
- *Linking rule*: function of the current status of the environment that associates to each node a neighbourhood.
- *Neighbourhood*: an entity composed by a node and a set of nodes, the neighbours.
- *Reaction*: any event that can change the status of the environment.
- *Condition*: a function that takes the current environment as input and returns a boolean and a number. The boolean value define if the reaction has to be taken, the number influences the reaction speed.
- *Action*: models an environment change.

More details about Alchemist can be found in its official documentation [1].

[2] https://github.com/AlchemistSimulator/Alchemist

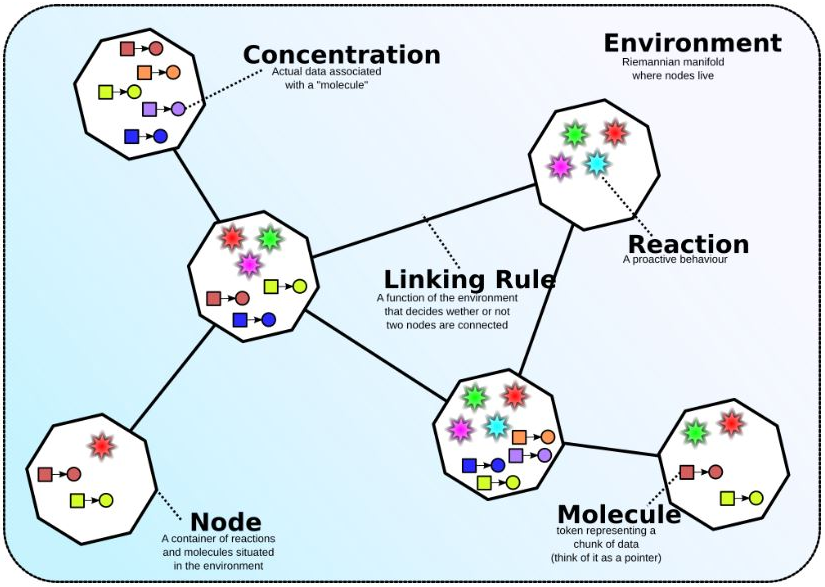


Figure 2.3: Computational model of Alchemist [19].

Incarnation

The framework presents a modular and extensible architecture. An Alchemist *incarnation* includes a type definition of concentration and possibly a set of specific conditions, actions and (rarely) environments. In other words, an incarnation is a concrete instance of an Alchemist meta-model [19]. The standalone distribution comes with: *Protelis*, *Sapere*, *Biochemistry* and *Scafi* [1].

Simulation

In good order to write a simulation is necessary to define a specific incarnation and fully describe environment and reactions, using *YAML language*.

2.5 Aggregate Computing and Security

One of the topics of interest related to Aggregate Computing is computer security. In compliance with [5], Aggregate Computing systems are vulnerable

to security threats due to their distributed nature, situatedness and openness, which can make participant nodes leave and join the computation at any time.

The attack surface is very large; in fact attacks may hit:

- *Infrastructure*: the physical environment may be manipulated and/or the communication network interrupted.
- *Physical devices*: the hardware, sensors and actuators may be altered or hijacked.
- *Platform level*: databases may be sabotaged and fake messages can be sent.

Despite this, the capacity of Aggregate Computing to abstract from the system's execution strategy can provide resistance to certain types of attack, or anyway can provide to the programmer the possibility of make design choices that take into account the security of the system. As a matter of fact, infrastructural malfunctioning could be avoided by adjusting interaction and computation consistent with communication media and endpoints. For the logical design instead, Aggregate Computing promotes decentralised systems and ad-hoc networking where computation is physically distributed and peer-to-peer interactions happen according to physical proximity, avoiding single points of failure typical of central servers or cloud endpoints.

From a functional point of view, the decentralised nature and the peculiar characteristics of adaptivity of Aggregate Computing approach make aggregate applications resilient to intermittent or prolonged failure of some nodes. However, the actual impact of node malfunctioning depends on many factors, such as the topology of the network, its density and the role that a node plays in the application [5].

2.5.1 Related work

In [5] the authors work on attacks to aggregate systems at application-level, where they assume that nodes are untrusted and, potentially, can create and

inject fake messages that can be received and used by their neighbourhood. In particular the fake messages can be: *malformed* or *well-formed*. For malformed messages there are no heavy problems because the Aggregate virtual machine ensures that the nodes which emit this type of messages are automatically ignored. Instead, if well-formed messages with malevolent payload are broadcasted, they can make the nodes decide erroneously and potentially compromise the entire network spreading those malicious information.

This problem arises due to the cooperative nature of the aggregate applications, in fact in aggregate systems each device has to appropriately participate to the calculation process and at the same time it has to maintain autonomy with respect of mobility, sensing and actuation. For these reasons an appropriate security strategy would require countermeasures to be applied across the aggregate computing stack.

Trust Framework

Typically the trusted relations are based on direct observations and advises gathered interacting with the neighbourhood. So once a trust metric is defined, the trust-base decision-making policy is responsible of comparing the trust estimated by a node, called *trustor*, with the expected behaviour of another node, called *trustee*, and a trust threshold value. In current literature are presented several solutions, many of which are based on the Bayesian approach, where the trustor uses an unknown parameter θ to predict a probabilistic future behaviour of the trustee. There are several probability prior distributions solutions, particularly in [5] the *beta distribution* is employed together with an ageing mechanism applied to the Aggregate Computing. In the beta distribution there are two parameters, α and β, which count the number of positive and negative observations experienced by the trustor while interacting with the trustee; the evaluation of each observation depends from the context and is formally computed:

$$E(Beta(\alpha + 1, \beta + 1)) = \frac{\alpha + 1}{\alpha + \beta + 2} \tag{2.1}$$

where α and β define the quality of the information that the nodes share at each round to define their trustworthiness. More specifically, the quality estimate is based on the calculation of the gradient, and the node is considered trust if for each node the value is similar to the neighbourhood values. Otherwise, when there are obvious perturbations the node is considered fake. The code of [5] is the following:

```
rep(Double.PositiveInfinity){ distance =>
  val dist = if(!fake) distance else fakeValue
  def nbrDist = nbr{dist}
  val n = countHood(nbrDist.isFinite)
  val sumValues = sumHood(mux(nbrDist.isFinite){nbrDist} {0.0})
  val xmean = sumValues/n
  val sumSqDev = sumHood( mux(nbrDist.isFinite)
    {Math.pow(dist-xmean, 2)} {0.0})
  val s = Math.sqrt(sumSqDev / n)
  mux(source) {0.0} {
    val res = foldhoodPlus(Double.PositiveInfinity)(Math.min){
      val trustParams = calculateTrustParams(dist, xmean, s)
      val trustValue = beta(trustParams.a, trustParams.b)
      val isTrusted =
        if(trustParams.numObservations >= minObservations)
          trustable(trustValue)
        else true
      val nbrId = nbr{mid()}
      if(!isTrusted) {distrustedNbrs=nbrId :: distrustedNbrs}
      val  newg = mux(isTrusted)
        {nbr{dist}+nbrRange}{Double.PositiveInfinity}
      newg
    }
    res.orIf(_==Double.PositiveInfinity){dist}
    {minHood( nbr{dist} + nbrRange )}
  }
}
```

Listing 2.1: Trust gradient implementation.

It emerges that the structure used is that of a classic gradient where, moreover, for each node:

- before and after the `mux` field, a data collection is performed. It is useful for the purpose of verifying whether a node is considered trust or not;
- the `trustValue` is calculated using beta distribution;

- if `trustValue` $\geq$ `trustThreshold` and the number of neighbours is sufficiently high a node is considered *trust*. In this case the gradient is normally calculated;
- otherwise, the node is considered as *fake*. In this case the node should be ignored by the calculation of the gradient.

Starting from the work done in [5] we tried to implement a reinforcement learning algorithm that was capable of discriminate unsafe devices in aggregate computing systems. The work is outlined in the next chapter.

Chapter 3

Analysis

In this chapter we introduce our contribution. The development of the solution has been achieved with an iterative and incremental process where even the requirements have been progressively refined.

In order to maintain a correct engineering approach: in the first section we define the requirements and then we will analyse them. In the end we will discuss and analyse our problem.

3.1 Requirements

In this section we will underline some requirements that are considered fundamental.

3.1.1 Scala Library for Reinforcement Learning

1. The implementation of a Scala library that is able to capture known reinforcement learning algorithms.

   (a) This library must capture the main concepts of reinforcement learning: state, action, reward, Q-table, Q-function and policy.

   (b) It must be written in Scala and must extract as much as possible from the single case of study.

   (c) This solution must be flexibly applicable in various contexts.

2. The solution must be integrated and work with the Aggregate computing approach.

   (a) It must allow aggregate computations and calculations.

   (b) The interactions between the nodes are essential in order to produce results.

3. The solution must be tested with multiple episodes. For each episode it must be possible to view the computed Q-table.

3.1.2 Reinforcement Learning-based Trust Framework

1. The trust solution implemented in [5] must be used as a basis for the learning process.

   (a) It must be integrable into various application designs.

   (b) The solution should be improved by reinforcement learning.

2. Like in the trust solution, the reinforcement learning solution must be capable of identifying and discriminating fake nodes.

3.2 Requirements Analysis

We will now analyse the above mentioned requirements.

First of all, it is immediately obvious that in order to implement and test our solution it is necessary to use a simulator capable of supporting the Aggregate Computing approach. So, from a structural point of view, the system is shaped by a network of devices (or nodes) that stand in the same environment; the single node behaviour emerges from the global environment interactions. This means that, each individual node:

- interacts with its neighbourhood by transmitting its value (or state) and receiving messages from its neighbours;

- can sense and act on its environment by its sensors and actuators.

Given the observations, the reinforcement learning algorithm used by us must be capable of capture learning only when the entire network has finished to calculate its state, because we are interested in the computation of the community, not of the individual. So it has been immediately clear that we had to work in a multi-agent reinforcement learning (MARL) context.

There are several reinforcement learning algorithms, they all require the right manipulation of environment concept and agent, consequently everything that was defined in the requirements. However these algorithms differ from each other for:

- knowledge of the environment: what can be totally known, partially known and totally unknown;
- immediately and long round reward processing;
- Q-table and policy updating.

For better clarity and understanding it is required to save the results obtained during each learning episode.

Instead, regarding the trust solution, it must be reorganised and integrated as required. For greater abstraction, we decided to encapsulate the trust solution within our global solution.

For identifying and discriminating fake nodes, through aggregate computations is necessary to identify those nodes which send perturbed values to their neighbourhood:

- Check which nodes send different values.
- Ignore these values during subsequent processing.

3.3 Problem Analysis

Working in an aggregate environment where, by definition, different devices exist and cooperate in the same environment, it is obvious that the environ-

ment is partially known and therefore each agent is able to recognise and communicate only with its own range of neighbours. Model-based learning solutions can hence be excluded, and model-free approaches begin to be explored.

After a careful analysis of the literature, we decided to use the Monte Carlo approach as reinforcement learning algorithm. This choice was taken because we needed the calculation of the result to be done only when an episode is declared concluded, in order to allow to capture the result obtained from the final interactions of the devices' network and therefore have an overall result. This can be seen as a TD(λ) approach where λ is not fixed; the policy is updated only at the end, using real sample results, but at the same time we incrementally try to improve the results on the basis of the previous optimal policy.

A very important technical issue to underline is the problem of the "alignment" in aggregate computing. In fact, the network devices can interact with each other only if they run the same program. But in a program devices only interact in common sub-computations. This is in line with the fact that we have global specifications which must lead to coherent collective behaviours.

Based on the trust algorithm, it seemed natural to maintain the computation of the gradient as a case of study. In particular, starting from one or more nodes defined as *source*, each nodes must:

- Calculate its distance from the source node. If more sources are present, the distance from the nearest one is only evaluated.
- Send its state to its neighbours.
- Receive neighbours' values.
- Evaluate through the trust framework if a given neighbour is reliable or not:
  - If it is trusted, the gradient is calculated
  - Otherwise is considered as a fake node.

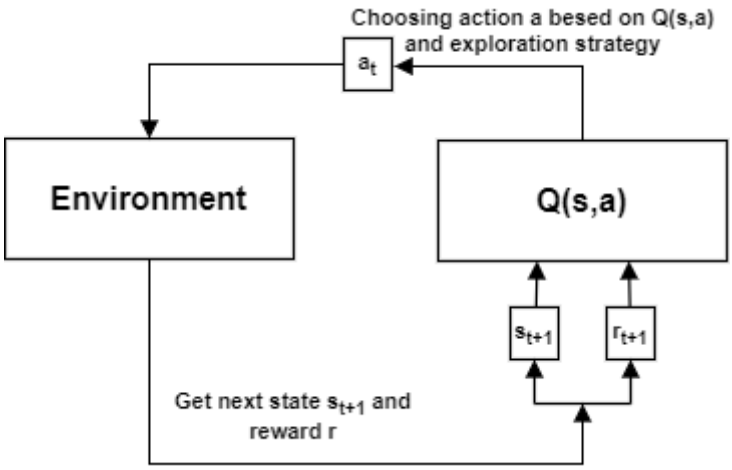


Figure 3.1: Reinforcement Learning interaction cycle. This represents the basic model from which our work started.

Starting from these first considerations it is now possible to define the learning model implemented. As shown in fig. 3.1 an agent which is in a given state selects an action (among those that can be performed in that state) based on the Q-table value and a given exploration strategy. This action produces environment changes, then the agent receives a reward and changes its state.

More specifically:

- The agent chooses an (s,a) pair.
- The agent receives a reward based on the (s,a) pair.
- The agent computes the gradient, evaluates neighbours for trust and changes its state.

After these observations we also define a use case (fig. 3.2) that allows us to have a global panoramic of our system.

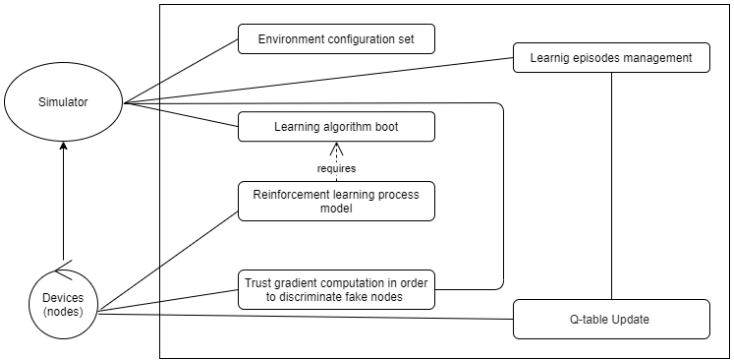


Figure 3.2: System use-cases diagram.

3.4 Domain Model: Overview

After a careful analysis of the requirements, four main components were immediately found (fig. 3.3):

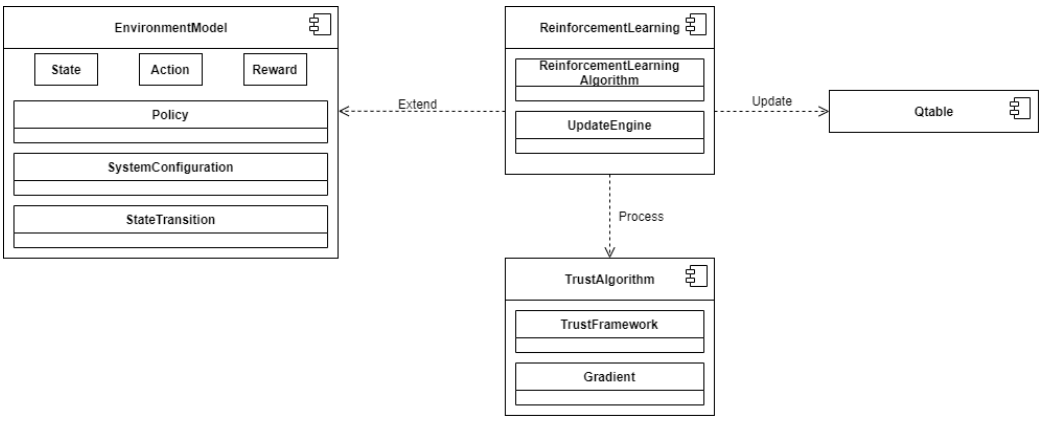


Figure 3.3: First system architecture design.

- `EnvironmentModel`: this component models the main reinforcement learning concepts and the environment.
- `ReinforcementLearning`: implements the reinforcement learning engine.
- `QTable`: manages Q-table update and storage.

- `TrustAlgorithm`: it contains the modified gradient code, which allows to identify and dismiss the fake neighbours. In other words it contains the trust algorithm.

Chapter 4

Design

At this point we are able to define a design model capable of describing, more or less in detail, how the implementation should be done. This chapter intends to describe the main elements of our application's design. At first a general design architecture is presented, then each component will be separately analysed in detail. This architecture has been projected in collaboration with thesis supervisors and starting from a prototype they supplied.

4.1 Design Architecture

Going in great detail the main components that make up our system are (fig. 4.1):

- `MCRL`: defines a trait of reinforcement learning.
- `MCRLImpl`: implements MCRL and characterises a learning core.
- `ManagementQtable`: set of functions to manage the Q-table.
- `GMCRL`: a singleton object useful to determine the learning engine.
- `MonteCarloLearning`: implements the logic of our experiment.
- `TrustAlgorithm`: encapsulates the trust algorithm presented in section 2.5.1.

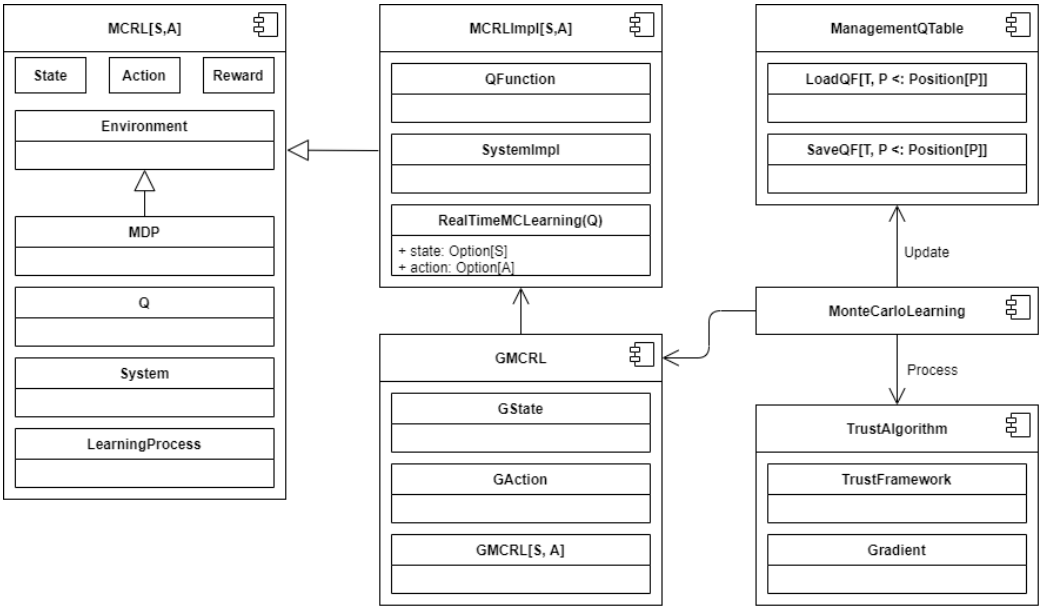


Figure 4.1: Design architecture.

Each individual component is described in depth below.

4.1.1 Reinforcement Learning Model

As defined in fig. 4.2, an attempt to define a reinforcement learning model, regardless of the algorithm used and capable of capturing all the common concepts of reinforcement, was made.

The `Q` trait extends the concept of `(S,A)` $\Rightarrow$ `R`, that corresponds to the reward associated to a given pair (state, action) and it is implemented by the class `QFunction`, a map-base implementation to manages Q function. In addition `Q` defines fundamental concepts of reinforcement learning, like the update table model, optimal value-function and possible exploration strategies. In particular the exploration strategy can be greedy or ϵ-greedy:

- greedy: selects action with highest value.
- ϵ-greedy: continues infinitely to explore and:
  - with probability $1 - \epsilon$ selects the action with the highest value;

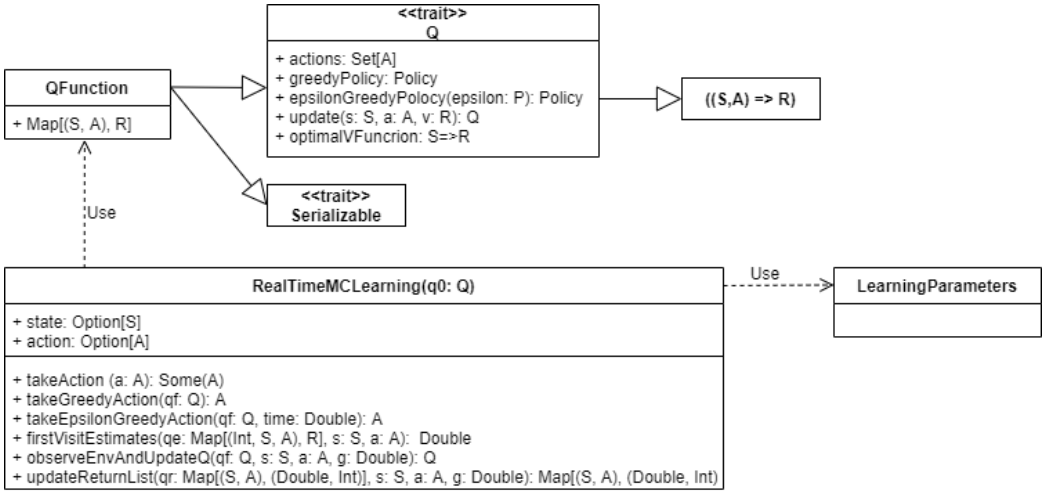


Figure 4.2: Reinforcement learning model design architecture.

– with probability ϵ selects a random action.

LearningParameters is a support class used here to define the constant variables employed in Monte Carlo, such as the *gamma* (discount value), *alpha* (learning rate) and *epsilon* parameters.

Finally, `RealTimeMCLearning` is an essential component for our implementation. This class provided a suite with methods needed for the reinforcement learning. In fact here we find:

- `takeGreedyAction`: implements the greedy exploration strategy;
- `takeEpsilonGreedyAction`: implements the ϵ-greedy exploration strategy;
- `observeEnvAndUpdate`: allows to update the values in the Q-table at the end of each episode. We implemented it as a map of the type: (state, action) $\rightarrow$ reward.

In this class we also defined some specifics of the Monte Carlo learning method, like an estimates rule and the relative rewards' processing.

4.1.2 Reinforcement Learning Engine

We present the library engine. As outlined in fig. 4.3, we created here the access point to the implementation of the reinforcement model and the main data structures useful to manage the algorithm. Furthermore, the definition of the different sets of state and action is made possible thanks to the `GState` and `GActions` traits.

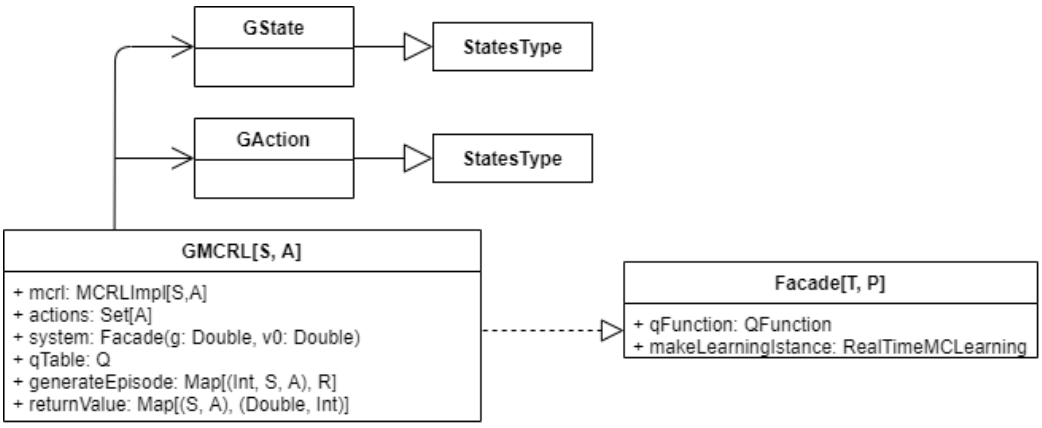


Figure 4.3: Reinforcement learning engine design architecture.

4.1.3 Monte Carlo Learning

In fig. 4.4 we want to capture the logic of the experiment. As already said, `GMCRL` allows us to access the engine and therefore the model of reinforcement learning introduced in section 4.1.2 and section 4.1.1. New concepts include:

- `RlBasedAlgorithm`: trait that defines the application logic.
- `RLBasedradient`: implements fundamental methods in order to execute our experiment. We defined here:
  - `Run`: computes the gradient using the methods defined according to the experiment. In our case it will initiate the trust gradient.
  - `Reward`: defines the immediate reward value, following a given rule.

  - `State`: defines the rule for assessing the current agent's state.

- `ManagementQTable`: encapsulates the update logic of the Q-table value and provides episodes' management support.

- `MonteCarloLearning`: the main class, which allows the experiment to be performed.

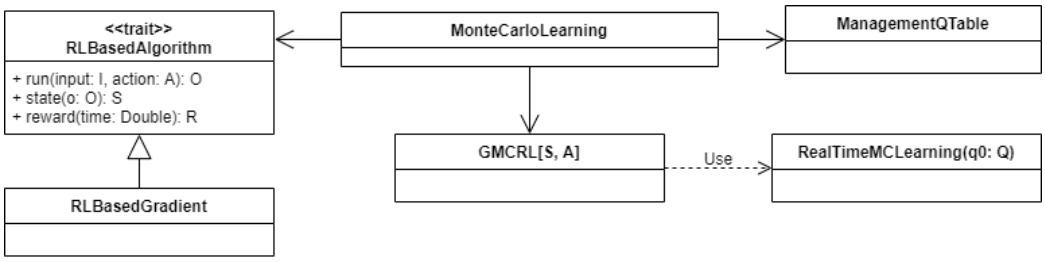


Figure 4.4: Monte Carlo learning design architecture.

4.2 The Overall System's Flow

The fig. 4.5 shows the overall system flow from the beginning of the application, through the creation of the Monte Carlo model and the reinforcement learning library as the result of the learning instance call, and ending with the episode termination.

First the application initialise the environment. Then that the Monte Carlo Learning can require to the reinforcement learning Scala library to create the useful to our model data structures like `qTable` or `genereteEpisode`. When the initialisation is done, the Monte Carlo model runs the reinforcement learning engine. Therefore, until the episode is over, each node:

1. set its state;

2. chooses an action following the exploration strategy;

3. it is awarded a reward based on the selected state-action pair;

4. updates the `genereteEpisode` structure adding the sequence (state, action) → reward;

5. process the new gradient calling the trust algorithm in order to detect the fake node. At the end it return the gradient value and it change the environment.

When this cycle is done, the application calls the Q-table update manager. For each state and action:

1. the first-visit estimates is calculated;

2. on the basis of the first-visit result and according to the learning rules the Q-table is updated.

This flow is repeated for each learning episode.

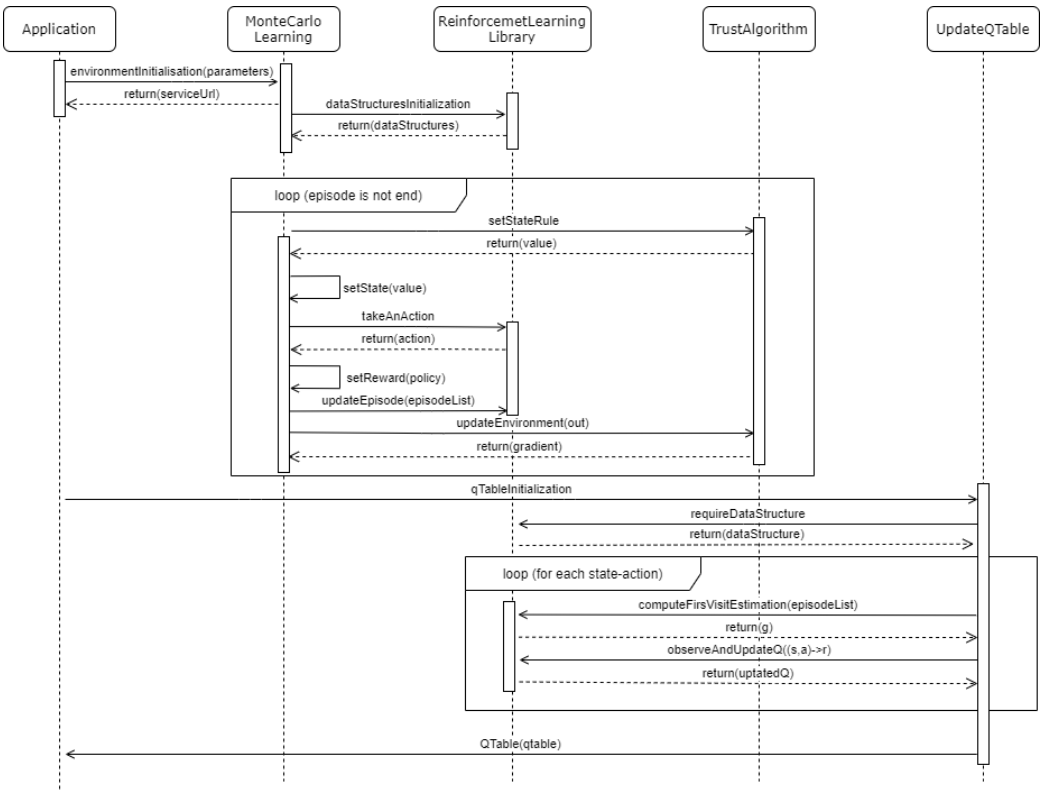


Figure 4.5: Overall system flow.

Chapter 5

Implementation

Based on the design model delineated in chapter 4 now we want to present our implementation in detail. Therefore, in this chapter we will specify the implementation choices, with references to code, which brought us to the creation of our library.

Outline:

- Project organization
- Reinforcement learning library implementation
- Monte Carlo learning implementation
- Application logic implementation
- Simulation

5.1 Project Organisation

For greater organisation, the project has been divided in packages as follows:

- `model`: implements the reinforcement learning based model and its functionality. In other words, it represents the core of our library.

- `trustAlgorithm`: includes the refactored work of [5].
- `update`: implements Q-table end episode management.
- `caseStudy`: contains example programs.

The project is versioned with *Git* and uses *Gradle Build Tool* for project and build automation.

5.2 Model Implementation

Starting from a prototype already provided by the supervisors, we have progressively defined the library model. The solution was implemented in Scala[1] using a few features and techniques described in chapter 1 and chapter 2. This allowed us to define a flexible Scala library for reinforcement learning applicable to Aggregate Computing contexts.

5.2.1 Generic Reinforcement Learning Model

First, an interface to manage the learning model has been provided. It has the task of defining the main concepts of the learning process, abstracting from the selected algorithm type.

Particularly, as specified in section 1.2, we define:

- `Policy`: delineates the mapping from the perceived states of the environment to the actions that can be taken.
- `VFunction`: identifies the best cumulative total reward for a given state.

  The V-function and the policy must be optimised and therefore it is necessary to identify the optimal policy and optimal value-function.
- *Exploration Strategy*: what can be greedy or ϵ-greedy. Where:
  - a greedy strategy privileges exploitation;

[1] https://www.scala-lang.org/

– an ϵ-greedy strategy privileges exploration.

For a better balance between exploitation and exploration (tackled in section 1.2.2) it is necessary to use a strategy that initially prefers exploration and subsequently, as time passes, it prefers exploitation.

- `update`: defines Q-table pattern.

Fundamental is the definition of *generic types* that allow us to map the definition of `State`, `Action` and `Reward` respectively like: `S`, `A`, `R`.

The above observations brought to the drafting of the following code:

```
type Policy = S=>A  // A strategy to act
type VFunction = S=>R // A state-value function

trait Q extends ((S,A)=>R) {
    def actions: Set[A]
    def greedyPolicy: Policy = s => actions.maxBy{this(s,_)}
    def epsilonGreedyPolicy(epsilon: P): Policy = {
        case s if Stochastics.drawFiltered(_<epsilon)
                => Stochastics.uniformDraw(actions)
        case s => greedyPolicy(s)
    }
    def update(s:S, a:A, v: R): Q
    def optimalVFunction: VFunction = s => actions.map{ this(s,_) }.max
}
```

Listing 5.1: Q trait implementation.

This trait is implemented by `QFunction` case class in order to create a correct function to manipulate the reinforcement process.

Then, we proceeded with the implementation of a class that allowed the management of the various reinforcement learning steps. In fact during the computation of an episode, an agent for each step: (*i*) sets its status based on observations made on the environment; (*ii*) selects an action based on the given exploration strategy; (*iii*) update the Q-table values. Notice that the time instant in which this last step occurs depends on the type of the algorithm used.

To implement this procedure the following code has been outlined:

```
case class RealtimeLearning(gamma: Double, q0: Q) {
```

```
    private var state: Option[S] = None
    private var action: Option[A] = None

    def setState(s: S) {
      state = Some(s)
    }
    def takeAction(a: A) {
      action = Some(a)
    }
    def takeGreedyAction(qf: Q): A = {
      val a = qf.greedyPolicy(state.get)
      takeAction(a)
      a
    }
    def takeEpsilonGreedyAction(qf: Q, time: Double): A = {
      val epsilon = epsilonDecayWithTime(time)
      val a = qf.epsilonGreedyPolicy(epsilon)(state.get)
      takeAction(a)
      a
    }
    def observeEnvAndUpdateQ(qf: Q): Q = {
      qf.update(state, action, v)
    }
}
```

Listing 5.2: Main methods of reinforcement learning.

It is notable that in the `takeEpsilonGreedyAction` method, in order to reach a exploration-exploitation trade-off, *epsilon* is used as a function of time (`epsilonDecayWithTime(time)`). More details will be provided in Learning Parameters section 5.2.1.

Learning Parameters

In order to improve the experience, for the definition of the learning support parameters, an object has been implemented, with the additional parameters: *learning rate* α, *exploration rate* ϵ and *discounting rate* γ.

While for the discounting rate literature prefers stable values, like 1 (non-discounted) or 0.99 (discounted), for α and ϵ the discussion is still open. We have decided to give a double possibility: calculate them as a function of time t or as a fixed value.

```
case object RLParameters {
    lazy val epsilon = 0.3
    lazy val alpha = 0.5
    def epsilonDecayWithTime(t: Double) =
        Math.min(EPSILON_MAX_VALUE, EPSILON_MIN_VALUE +
        (1-EPSILON_MIN_VALUE) * math.exp(-DECAY_VALUE * t))
    def alphaDecayWithTime(t: Double) =
        Math.max(MIN_VALUE, Math.min(ALPHA_MAX_VALUE,
        ALPHA_MIN_VALUE - Math.log10((t+1)/1000.0)))
}
```

Listing 5.3: Reinforcement learning parameters implementation.

To enhance the trade-off between exploration and exploitation ϵ it is recommended to use `epsilonDecayWithTime(t: Double)`. The method allows the decrease of the exploration factor and the speed of this decreasing is given by the `DECAY_VALUE` variable.

As regards, α it behaves similarly to ϵ but learning rate determines how much the new acquired information have to extent and/or overwrite the old ones.

5.2.2 Generic Reinforcement Learning Engine

We will now talk about the access point to our library. A facade class is needed to simplify the initialisation of all the data structures, useful to keep track of the learning and to provide a single, coherent and consistent access point for each node of the network.

For this purpose the class `Facade[T,P](gamma:Double, v0:Double)` has been developed, to allow us to create a `qFunction` and an instance of the learning process (`makeLearningInstance`).

In the end a `qTable` variable is built, to allow to keep track and access the values entered in the table. In particular, the Q-table allows us to capture the best reward associated to each state-action pair.

```
class GMCRL[S,A](
                val mcrl: MCRLImpl[S,A],
                val actions: Set[A]) {
    case class Facade[T,P](gamma: Double,
                           v0: Double) {
```

```
        import mcrl._
        def qFunction = QFunction(actions, v0)
        def makeLearningInstance() = RealtimeLearning(gamma, qFunction)
    }
    val system = Facade(gamma = 1, v0 = 0.0)
    val mcLearning = system.makeLearningInstance()
    var qTable = mcLearning.q0
}

object GMCRL extends GMCRL[GState, GAction](
                            new MCRLImpl[GState,GAction]{ },
                            Set[GAction](<action_name>)) {
    val states = Set[GState](<state_name>)
}
```

Listing 5.4: Reinforcement learning engine implementation.

States and Actions

The definition of states and actions selectable by agents at any instant t follows this pattern:

```
trait GState
case object <state_name> extends GState
...
trait GAction
case object <action_name> extends GAction
...
```

Listing 5.5: State and action implementations.

5.2.3 Monte Carlo Implementation

In accordance with the Monte Carlo algorithm, presented in section 1.5.1, we have developed the main concepts for its proper functioning. Recalling that Monte Carlo employs averaging sample returns for episodic tasks, and therefore updates the values in the Q-table only when an episode is done. So, at each episode, the node i performs the following operations:

1. Choose state-action pair, initially randomly, then following the values in the Q-table.

2. Generate an episode following the policy.

3. When the episode ends:

   (a) if the value is present in the episode, calculate the g value, according to the first-visit rule, and add it to Returns (s, a);

   (b) update the Q-table;

   (c) update the policy.

Particularly, has been necessary to add two new data structures to section 5.2.2:

- `generateEpisode`: variable that keeps track of the reward associated to each action-state pair selected at each instant t in an episode. This is modelled by an `LinkedHashMap[(Int,S,A), Double]` in order to maintain the insertion order.

- `returnValue`: maintains the total of the return values and the number of total visits for each state-action pair. This variable is updated when an episode is done by: `updateReturnList(qr: Map[(S,A), (Double,Int)], state: S, action: A, g: Double)`.

```
def updateReturnList(
                    qr: Map[(S,A), (Double,Int)],
                    state: S, action: A, g: Double) = {
    if(qr.contains(state,action))
        qr += ((state, action) -> (qr(state,action)._1 +
                g, qr(state,action)._2+1))
    else qr += ((state, action) -> (g, 1))
}
```

Listing 5.6: Return list implementation.

First-visit

As discussed in section 1.5.1, we implemented first-visit, that we remember to estimate q(s,a) as the average of the returns for the first visit to (s,a), as following:

```
def firstVisitEstimates(
                        qe: LinkedHashMap[(Int,S,A), R],
                        state: S, action: A ) = {
    val firstOccurrence = qe.find(
        v => v._1._2 == state && v._1._3 == action)
    while (qe.head != firstOccurrence.get) qe-= qe.head._1
    qe
}
```

Listing 5.7: First-visit estimates implementation.

Computing of G Value

Once the first-visit has been processed, we calculate the `G` value for each state-action pair in the following way:

```
def updateG(
            qe: LinkedHashMap[(Int,S,A), R],
            state: S, action: A): Double = {
    val episodes = qe.clone()
    val tmp = firstVisitEstimates(episodes, state, action)
    val g = tmp.map{case((t,_,_),v) => v*math.pow(gamma, t.toDouble)}.sum
    g
}
```

Listing 5.8: Computing of the G value implementation.

where γ is the discounted rate and $0 \leq \gamma \leq 1$.

Q-table Update Value

At the end we proceed with the updating of the values in the Q-table. For this we modified the method defined in section 5.2.1 in the following way:

```
def observeEnvAndUpdateQ(
                    qf: Q,
                    returnValue: Map[(S,A), (Double,Int)],
                    a: S, a: A, g: Double): Q = {
    /* Incremental solution
     * val ns = 1/returnValue(s,a)._2
     * val v = qf(s,a) + ns * (g - qf(s,a))
     */

    val v = qf(s,a) + alpha * (g - qf(s,a))
    qf.update(s, a, v)
```

```
}
```

Listing 5.9: Updating of Q-table implementation.

Notable that there are two methodologies to calculate the v value:

- $v = qf(s,a) + ns * (g - qf(s,a))$
  Averaging sample returns.
- $v = qf(s,a) + alpha * (g - qf(s,a))$
  In non-stationary problems, it can be useful to track a running mean.

All of those methods were added in the class `RealtimeLearning` (section 5.2.1).

5.3 Application Logic Implementation

Analysing the application logic, we decided that each node of the network must:

1. set its state;
2. choose an action;
3. receive a reward to associate with a state-action pair;
4. change the environment according to the defined rules.

This is repeated for each temporal instant t, where $t = 0, 1, \ldots, T$ and T is the end of an episode.

In order to capture this cycle a method accessible by all the nodes of the network was necessary to allow us to define for each episode the sequence of `(state, action)` $\rightarrow$ `reward` captured at each instant t, implemented in the following way:

```
def monteCarloLearning[I,O,A,S](
                algorithm: RLBasedAlgorithm[I,O,A,S,Double],
                mc: GMCRL[S,A],
                o0: O, a0: A, input: I, t: Double,
```

```
                learn: Boolean = true): O =
    rep((o0,a0)) { case (o,a) =>
      val action = branch(!learn){
        mc.mcLearning.setState(algorithm.state(o))
        mc.mcLearning.takeGreedyAction(mc.qTable)
      } {
        val state = algorithm.state(o)
        mc.mcLearning.setState(state)
        val action = mc.mcLearning.takeEpsilonGreedyAction(mc.qTable, t)
        val reward = algorithm.reward(action, t)
        mc.generateEpisode += (t.toInt, state, action) -> reward

        node.put("state", state)
        node.put("action", action)
        action
      }
      val output = algorithm.run(input, action)
      (output, action)
    }._1
```

Listing 5.10: Monte Carlo application logic implementation.

The calculation of the total estimate of the rewards and the update of the Q-table will be carried out later, when the episode ends. It will be described in detail in the section 5.5.

To make this sequence more reusable, an interface has been projected:

```
trait RLBasedAlgorithm[I,O,A,S,R] {
    def run(input: I, action: A): O
    def state(o: O): S
    def reward(time: Double): R
  }
```

Listing 5.11: Generic interface in order to capture learning cycles.

where: `I`, `O`, `A`, `S` and `R` defined the generic types which represents respectively the: input, output, action, state and reward.

This trait is instantiated through the class that will allow us to define the reinforcement rules based on the result we want to achieve.

5.4 Trust Algorithm

In this section we are going to talk in depth about the trust algorithm used by our application to identify and exclude the fake nodes from the calculation of the gradient.

The solution presented in section 2.5.1 has been used, which after a first phase of refactoring has been encapsulated in our experiment.

So, now the interface presented in section 5.3 will have a class that implements the `run` method by calling the trust algorithm.

Greater attention goes to the rules to identify and delineate a node as fake:

```
def gradientWithTrust(
                    source: Boolean,
                    fake: Boolean = false,
                    fakeValue: Double = 0.0): Double = {
   rep(Double.PositiveInfinity){ distance =>
     <trust_parameters>
     mux(source) { 0.0 } {
         <another_trust_parameters>
         val trustParams = calculateTrustParams(dist, xmean, s)
         val trustValue = beta(trustParams.a, trustParams.b)
         val isTrusted =
           if(trustParams.numObservations >= minObservations)
               trustable(trustValue)
           else true
         if(!isTrusted)
           {distrustedNbrs = nbrId :: distrustedNbrs}
          val newg = mux(isTrusted)
           { nbr{dist} + nbrRange }{ Double.PositiveInfinity }
           newg
         }
         <add_untrust_node_in_list_of_distrusted>
         res.orIf(_==Double.PositiveInfinity){dist}
     }
   }
}
```

Listing 5.12: Trust algorithm implementation.

The beta function distribution calculates the beta distribution, this value is then passed to the `trustValue`. With `isTrusted`, instead, we define if a node is to be considered trust or not, after evaluating it using a threshold:

`trustThreshold`.

```
def beta(a: Double, b: Double) = (a+1)/(a+b+2)
def trustable(trustValue: Double): Boolean = trustValue >= trustThreshold
```

Listing 5.13: Trust parameters implementation.

On the basis of these choices the gradient is calculated taking into account the neighbour's value.

5.5 Update Q-table Implementation

In this section we examine the last phases: an episode ends and the Q-table must be updated. An ad-hoc class has been created to capture the termination of an episode. For each state- action we call the methods defined in section 5.2.3. After updating the table, the results can be printed.

Here is the code that does these operations:

```
override def execute(): Unit = if(episode>0 && episode%saveEvery==0){
    var g = 0.0
    for(s <- GMCRL.states;
        a <- GMCRL.actions){
        val condition = GMCRL.generateEpisode.exists{
            case((_,state,action),_) => st.equals(s) & act.equals(a)}
        if(condition) {
            g = GMCRL.mcLearning.updateG(GMCRL.generateEpisode, s, a)
            GMCRL.returnValue =
               GMCRL.mcLearning.updateReturnList(GMCRL.returnValue, s, a, g)
        }
        GMCRL.qTable =
            GMCRL.mcLearning.observeEnvAndUpdateQ(GMCRL.qTable, s, a, g)
            // insert GMCRL.returnValue, for classical computing of value
    }
    ...
    for(s <- GMCRL.states;
        a <- GMCRL.actions){
      oos.writeDouble(GMCRL.qTable(s,a))
      bos.write(s"($s,$a) = ${GMCRL.qTable(s,a)}\n")
    }
    GMCRL.generateEpisode.clear()
    ..
  }
}
```

Listing 5.14: Implementation of Q-table update and management.

The latter class is essential to start our analyses and start the *batch-execution* of our solution.

5.6 Simulations

As simulation environment to test our prototype we used ScaFi and Alchemist (presented respectively in section 2.3 and section 2.4). This allow us to comply with the requirement of the solution must be integrated and work with the Aggregate computing approach.

We decided to use Alchemist as a simulator to improve the performances, using a solid and performing platform for aggregate applications. Furthermore it offers flexibility and superior simulation control compared to an ad-hoc simulator.

It is important to underline the fact that, in order to perform multiple learning processes (multiple episodes), execution takes place in *batch*. The graphical interface, made available by Alchemist, allows us to capture a single episode.

Our Alchemist simulation is started by:

```
episodes: &episodes <num_of_episodes>

# Definition of variables
variables:
  ...
  minObservations: &minObservations
    formula: 6
  trustThreshold: &trustThreshold
    min: 0.40
    step: 0.05
    max: 0.95
    default: 0.8
 ...
incarnation: scafi
 ...
network-model:
  type: ConnectWithinDistance
  parameters: [*commRadius]

pools:
```

```
  - pool: &program
      - time-distribution: 1
        type: Event
        actions:
          - type: RunScafiProgram
            parameters: [it.unibo.monteCarloWithES.TrustOrClassic, 5.0]
      - program: send
    ...
  # Manages Q-table loads and saves
  - pool: &saveQF
      - time-distribution:
          type: Trigger
          parameters: [120.0]
        type: Event
        actions:
          - type: it.unibo.monteCarloWithES.update.SaveQF
            parameters: [qf]
 ...
# Network configuration
displacements:
  - in:
      type: Grid
      parameters: [0, 0, 10, 10, 1, 1, 0, 0]
    ...
```

Listing 5.15: YAML file with configuration in order to launch the experiments.

5.6.1 Application Flow

In fig. 5.1 we want to capture the flow followed by our prototype.

A YAML file to start the simulation has been written, it will start the main program. Then, we will instantiate the main structures that will enable the use of the reinforcement learning library and the trust algorithm will be launched. When an episode is done, all the operations that lead us to the calculation of the estimate are performed and the Q-table is updated. At the end the results are stored.

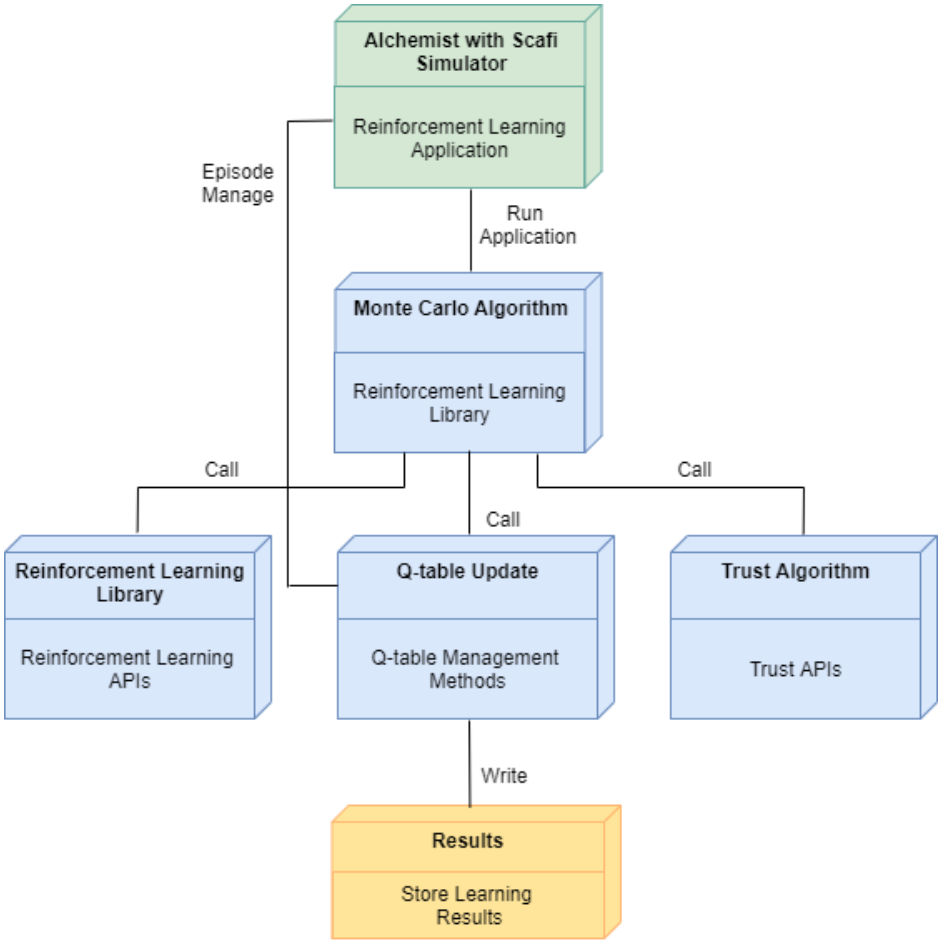


Figure 5.1: Application flow diagram.

5.7 How reinforcement learning fits aggregate computing

In fig. 5.2 is shown how the reinforcement learning based solution fits into the aggregate system dynamic rounds and how the state of the network evolves.

Starting from an initial configuration, where there are a source and fake node, the computation goes through those steps:

1. the source node and its neighbours compute the reinforcement learning based solution, in particular:

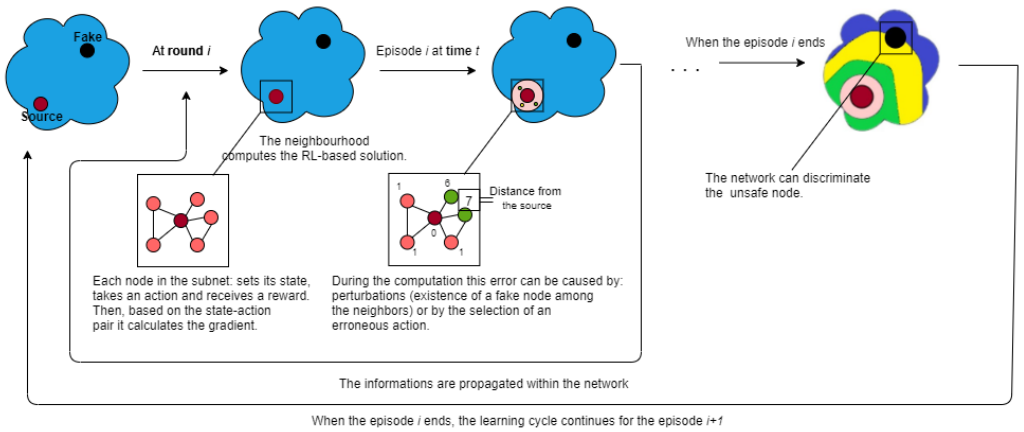


Figure 5.2: Reinforcement learning fits in aggregate computing dynamics.

(a) each node of the sub-net sets its state, selects an action and receives a reward;

(b) based on the state-action pair selected computes a gradient,

(c) sets the environment and sends its value to the neighbourhood.

2. when the propagation of the information reaches each device in the network, through the round execution typical of the aggregate computing systems, the episode ends and the results are outputted;

3. the episode is incremented and the computation restarts from point 1.

Chapter 6

Evaluation

This chapter intends to evaluate how much the result of our application meets our requirements.

Therefore, we will discuss the requirements satisfaction. Then, we will present the case of studies and finally the results obtained will be examined.

6.1 Requirements satisfaction

Considering the requirements defined in section section 3.1, we evaluate which have been respected and which are to be improved.

6.1.1 Scala Library for Reinforcement Learning

- *Requirement 1*: our Scala library, by definition written in Scala, capture all the main concepts of reinforcement learning algorithm effectively abstracting from the type of algorithm used (ref. section 5.2.1).

- *Requirement 2*: each node of the network executes the program, selecting actions and calculating its reward and then writes it in an aggregate way. Furthermore, the reward is assigned according to a community condition and the gradient is calculated on the basis of the neighbour's value (ref. section 5.3 and section 6.2.1)

- *Requirement 3*: for each episode, the value associated with each state-action pair is stored according to the policies determined by the algorithm used (ref. section 5.5).

6.1.2 Reinforcement Learning-based Trust Framework

- *Requirement 1*: our solution encapsulates and integrates with the solution presented in [5] (ref. section 5.4). However a significant improvement has not been found, so on this point we keep ourselves open to future developments and improvements.

- *Requirement 2*: all our case of studies are able to identify fake nodes (ref. section 6.2 and section 6.3).

6.2 Case Study: Trust-based Gradients

A few demonstrative programs have been written in order to show how the library can be used to develop reinforcement learning program in order to discriminated unsafe devices in aggregate computing system.

6.2.1 Simulation Setup

All solutions use:

1. a network consisting of 100 nodes;

2. Monte Carlo Exploring Start and first-visit estimated (presented in section 5.2.3);

3. the trust algorithm (section 5.4), with these parameter values (among the many present):

   (a) `minObservations = 6`

   (b) `trustThreshold = 0.8`

4. discount rate: $\gamma = 1$;
5. reward rule.

The reward rule defines that:

- a positive reward is given if the value of the gradient of each node of the network is equal to the value that would have performed a classic gradient for at least n steps;
- otherwise a negative reward will be assigned. The negative reward increases when the time passes, in order to define the convergence times of the solution.

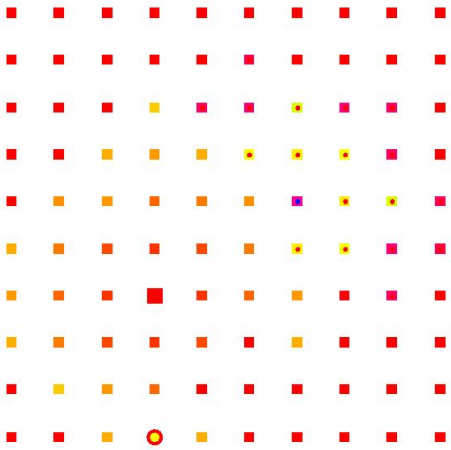
The following code captures this idea:

```
def isStableForEachNode = {
      for (n <- alchemistEnvironment.getNodes.iterator().asScala) {
        val classicValue = n.getConcentration(
            new SimpleMolecule("classicGIgnoreFake")).toString.toDouble
        val rlValue = n.getConcentration(
            new SimpleMolecule("rlbasedG")).toString.toDouble
        if (n.getId != FAKE_DEVICE) {
          rewardMap.put(
                        n.getId,
                        recentValues(STEP_EVALUATION, classicValue
                    ) ++ recentValues(STEP_EVALUATION, rlValue))
        }
      }
      rt = rewardMap.forall {  case (id, v) =>
                                rewardMap(id).forall(e => e == v.last) }
}
```

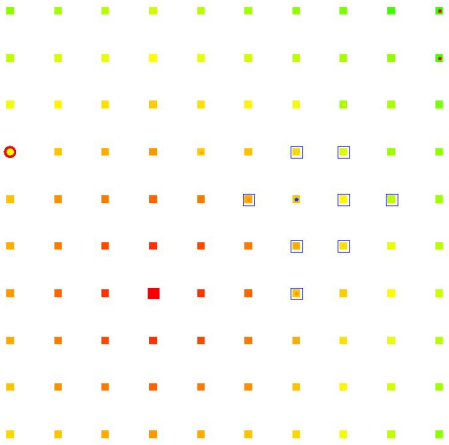
Listing 6.1: Reward rule implementation.

6.2.2 Graphical Evolution of the Simulation

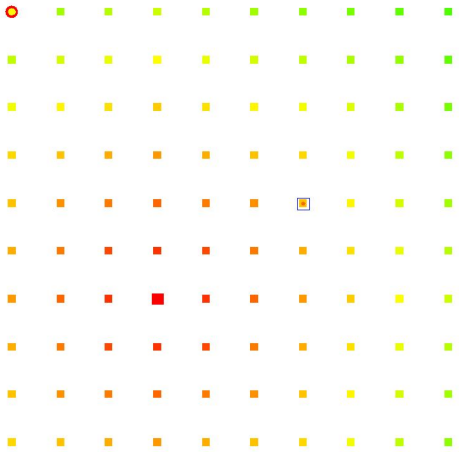
As shown in fig. 6.1, when a simulation is launched the gradient field computation begins. At first, the gradient field is calculated starting from both the source node and the fake node, which sends random values to its neighbourhood (fig. 6.1a). This is because the number of neighbours participating



(a) Step 1: gradient calculation.



(b) Step 2: the nodes understand that the gradient is corrupted.



(c) Step 3: fake is detected.

Figure 6.1: Phases of the experiment. In (a), the calculation of the gradient begins. The fake node has already started to infect. In (b) the nodes start to understand that the gradient is corrupted; then they start exchanging information about the nodes to be considered distrusted. In (c) fake is detected and is distrusted. At this point the gradient field is recovered.

in the distrusted valuation is still too low. Then, when the neighbourhood reaches an appropriate number, they understand that the gradient is corrupted; so the nodes exchange information in order to identify the fake node.

Remember that a node is considered false when its gradient value is too far from the values sent by its neighbours. In our simulation the nodes that are considered distrusted are selected by a blue rectangle (fig. 6.1b). At the end, when the algorithm identifies the fake node, it is excluded and labeled as distrusted. At this point the gradient field is recovered.

6.2.3 Solution 1: Trust or Classic Gradient

The goal of this simulation is to identify the gradient that takes less time to converge and therefore privileges its selection.

Configuration

- **State**: DummyState
- **Actions**: ClassicAction, TrustAction
- **Exploration strategy**: greedy
- **Update strategy**: averaging sample returns
- **Gradients**: TrustGradient, ClassicGradientWithFakeNode (or ClassicGradientIgnoreFake)

Implementation

In this solution, for simplicity, we have considered a single state. We have two possible actions, both selectable from the dummy state. One represents the execution of the classic gradient that does not ignore the neighbours while, the other, represents the execution of the trust gradient. Each node in the network must select the same action at a same instant t. The target is to train the network to select the action, and therefore the gradient, which takes less time to converge according to the rule defined in `isStableForEachNode`.

Code

```
// Set state and actions
trait GState
case object DummyState extends GState
trait GAction { def selection: Boolean }
class SelectionAlgorithm(val selection: Boolean = false) extends GAction
case object ClassicAction extends SelectionAlgorithm(true)
case object TrustAction extends SelectionAlgorithm(false)

// Set reinforcement configurations
class RLbasedGradient extends RLBasedAlgorithm
                [(Boolean, () => Double), Double, GAction, GState, Double] {
    override def run(input: GInput, action: GAction): Double = {
      val (source, isFake) = input
      if (action.selection) gradient(source, isFake, fakeValue)
      else gradientWithTrust(source, isFake, fakeValue)
    }
    override def state(o: Double): GState = {
      ClassicState
    }
    override def reward: Double = {
      val reward = if(isStableForEachNode) 0 else -time-1
      reward
    }
}
```

Listing 6.2: Solution 1: trust or classic gradient implementation.

6.2.4 Solution 2: Trust, Classic or Mix Gradient

Starting from solution 1, we asked ourselves how the prototype behaved by adding actions that involved the use of a mix gradient.

Configuration

- **State**: DummyState
- **Actions**: ClassicAction, TrustAction, MixAction
- **Exploration strategy**: greedy
- **Update strategy**: averaging sample returns

- **Gradients**: TrustGradient, ClassicGradientWithFakeNode (or ClassicGradientIgnoreFake), MixOfThePrevious

Implementation

In this solution we have only one state. We have five possible actions, all selectable from the dummy state. A φ value is defined; it represents the possibility of selecting one gradient with respect to another, according to the following rule:

$$\varphi * \text{Classic Gradient} + (1 - \varphi) * \text{Trust Gradient} \tag{6.1}$$

Each node in the network must select the same action at a same instant t. The target is to train the network to select the action, and therefore the gradient, which takes less time to converge according to the rule defined in `isStableForEachNode`.

Code

```
// Set state and actions
trait DummyState
case object ClassicState extends GState
trait GAction { def selection: Double }
class SelectionAlgorithm(val selection: Double = 1.0) extends GAction
case object ClassicAction extends SelectionAlgorithm(1.0)
case object MixAction1 extends SelectionAlgorithm(0.8)
case object MixAction2 extends SelectionAlgorithm(0.5)
case object MixAction3 extends SelectionAlgorithm(0.2)
case object TrustAction extends SelectionAlgorithm(0.0)

// Set reinforcement configurations
class RLbasedGradient extends RLBasedAlgorithm
                [(Boolean, () => Double), Double, GAction, GState, Double] {
    override def run(input: GInput, action: GAction): Double = {
      val (source, isFake) = input
      action.selection * gradient(source,isFake,fakeValue) +
        (1-action.selection) * gradientWithTrust(source,isFake,fakeValue)
      /* If the solution with ClassicGradientIgnoreFake is analysed
       * action.selection * branch(isFake){fakeValue}{gradient(source)} +
        (1-action.selection) * gradientWithTrust(source,isFake,fakeValue)
       */
```

```
    }
    override def state(o: Double): GState = {
      ClassicState
    }
    override def reward: Double = {
      val reward = if(isStableForEachNode) 0 else -time-1
      reward
    }
}
```

Listing 6.3: Solution 2: Trust or classic or mix implementation.

6.2.5 Solution 3: Action Selection Based on `isTrusted` Value

The goal of the experiment is learning to select the correct action basing on the state in which the agent is.

Configuration

- **State**: TrustState, NotTrustState
- **Actions**: ClassicAction, IgnoreAction
- **Exploration strategy**: greedy or ϵ-greedy
- **Update strategy**: averaging sample returns or non-stationary
- **Gradients**: TrustGradient, ClassicGradientIgnoreFake

Implementation

In this solution we have two possible states. The state is determined based on the `isTrusted` value, that is: if `isTrusted` is true the state will be set to `TrustState`, otherwise it is `NotTrustState`. We have two possible actions, both selectable from all states. The first involves running the classic gradient that ignores fake nodes (`ClassicAction`), the other involves the trust gradient (`IgnoreAction`). The expectation in this context is to have a reward that favours:

- ClassicAction when the state is TrustState;
- IgnoreAction when the state is NotTrustState.

Code

```
// Set states and actions
trait GState
case object TrustState extends GState
case object NotTrustState extends GState
trait GAction
case object ClassicAction
case object IgnoreAction

// Set reinforcement configurations
class RLbasedGradient extends RLBasedAlgorithm
                [(Boolean, () => Double), Double, GAction, GState, Double] {
    override def run(input: GInput, action: GAction): Double = {
      gradientWithTrust(source, isFake, fakeValue)
    }
    override def state(o: Double): GState = {
      if(distrusted) TrustState else NotTrustState
    }
    override def reward: Double = {
      val reward = if(isStableForEachNode) 0 else -time-1
      reward
    }
}
```

Listing 6.4: Solution 3: based on `isTrusted` value implementation.

6.2.6 Solution 4: Action Selection Based on `trustValue`

Starting from the `trustValue` variable (section 5.4) we want to learn how to select the right action.

Configuration

- **State**: SurelyIgnore, MaybeIgnore, DoNotIgnore
- **Actions**: Ignore, Gradient
- **Exploration strategy**: greedy or ϵ-greedy

- **Update strategy**: averaging sample returns or non-stationary
- **Gradients**: TrustGradient, ClassicGradientIgnoreFake

Implementation

In this solution we have three possible states. The state is determined based on the `trustValue`. This value is in the [0, 1] range, so we have divided it into slots. When the values are very low, the status will be set as `SurelyIgnore`, in intermediate cases such as `MaybeIgnore` and for high values `DoNotIgnore`. We have two possible actions, both selectable from all states. The first involves running the classic gradient that ignores fake nodes (`Gradient`), the other involves trust gradient (`Ignore`). Obviously the expectation in this context is to have a reward that favours:

- SurelyIgnore when the state is Ignore;
- DoNotIgnore when the state is Gradient.

Code

```
// Set states and actions
trait GState
case object SurelyIgnore extends GState
case object MaybeIgnore extends GState
case object DoNotIgnore extends GState
trait GAction { def selection: Double }
class SelectionAlgorithm(val selection: Double = 1.0) extends GAction
case object Ignore extends SelectionAlgorithm(1.0)
case object Gradient extends SelectionAlgorithm(0.0)

// Set reinforcement configurations
class RLbasedGradient extends RLBasedAlgorithm
                [(Boolean, () => Double), Double, GAction, GState, Double] {
    override def run(input: GInput, action: GAction): Double = {
      val (source, isFake, metric) = input
      gradientWithTrust(source, isFake, fakeValue)
    }
    override def state(o: Double): GState = {
      trustValue match {
        case v if v>= 0 && v<0.4 => SurelyIgnore
```

```
        case v if v>=0.4 && v<0.8 => MaybeIgnore
        case _ => DoNotIgnore
      }
    }
    override def reward(time: Double): Double = {
      val reward = if(isStableForEachNode) 0 else -time - 1
      reward
    }
}
```

Listing 6.5: Solution 4: based on `trustValue` implementation.

6.3 Results

The performances are evaluated through the accuracy index:

$$Accuracy = \frac{\text{Episodes completed successfully}}{\text{Total episodes}} \tag{6.2}$$

where correctly classified episodes are considered those that have a less negative cumulative reward for the state-action pair considered correct. Specifically, for each solution the table 6.1 shows the desired results.

Table 6.1: Legend of desired results for each experiments.

	Preference
Solution 1	DummyState-TrustAction
Solution 2	DummyState-TrustAction
Solution 3	TrustState-TrustAction NotTrustState-ClassicAction
Solution 4	SurelyIgnore-Ignore MaybeIgnore no preference DoNotIgnore-Gradiente

Note that, in *solution 1* and *solution 2*, if the simulation uses ClassicGradientIgnoreFake the DummyState-Classic solution is preferred.

The sampling was carried out on 100 episodes and the results are listed in table 6.2. Remember that as long as the algorithm does not identify the fake

nodes, the reward is negative and it increases as time passes. Furthermore, solution 1 (section 6.2.3) and solution 2 (section 6.2.4) converge after a few episodes and their accuracy is high. Although, solution 3 (section 6.2.5) takes longer to converge, the results are still good, in fact, after about 26 steps it stabilises. Finally, in solution 4 (section 6.2.6) the error rate is higher. Here the failure rate grows exponentially and the solution is quite unstable. In the few cases where the sampling takes place correctly, the values are very close to each other.

Table 6.2: Experiments results table.

	Exploration Strategy	**Accuracy**	**Avg Reward**	
Solution 1	greedy	99%	DummyState-ClassicAction	-3712.0
			DummyState-TrustAction	-250,47
Solution 2	greedy	95%	DummyState-ClassicAction	-53,9
			DummyState,MixAction1	-70,6
			DummyState-MixAction2	-250,47
			DummyState-MixAction3	-245,41
			DummyState-TrustAction	-253.0
Solution 3	ϵ-greedy	75%	TrustState-ClassicAction	-400,068
			TrustState-IgnoreAction	-533,068
			NotTrustState-ClassicAction	-532,566
			NotTrustState-IgnoreAction	-431,094
Solution 4	ϵ-greedy	35%	SurelyIgnore-Ignore	-512,995
			SurelyIgnore-Gradient	-533,257
			MaybeIgnore-Ignore	-570,358
			MaybeIgnore-Gradient	-570,394
			DoNotIgnore-Ignore	-524,74
			DoNotIgnore-Gradient	-514,70

Chapter 7

Conclusion

This chapter includes a few brief and general considerations about this thesis' work; as well as some references to the possible future developments.

7.1 Discussion

The reinforcement learning approach in the aggregate computing system discussed in chapter 1, chapter 2 and implemented in chapter 5 seems promising.

The prototype can learn the correct sequence of actions to take for basic cases of study. It can, for example, select the actions which favour the execution of a modified gradient in order to identify the fake nodes when there is an unsafe device in the neighbourhood. It prefers instead the execution of a classic gradient when the neighbours are considered trust. Good results have been obtained also when the selection of the gradient to execute is based on the state in which the agent is. The more the states and the actions increase the more certain problems emerge. As the matter of fact in those cases has been noticed that the agent has trouble to find the correct sequence of actions to choose and the failure rate increases. A more structured solution would be appropriate. In addition the reinforcement learning based solution is applied to a more "external" level then aggregate computing through the simulation. Therefore the learning is not done directly by the aggregate

system via collective computation.

However, this solution must be improved and made more performing in order to become more robust, both for different case of studies and for real world applications.

Although, we are aware of the big challenges and limitations that exist nowadays in the world of multi-agent reinforcement learning we believe that this work can be useful as a starting point for innovative solutions. Thanks both to the use of solutions that allow aggregate calculations and to reinforcement learning solutions. Allowing devices to learn information directly from data, without predetermined models, thus becoming a highly reusable solution. In fact, by modifying only a few variables it can be used in various application contexts.

7.2 Further Developments

Obviously our prototype is not complete. Below is a list with some possible developments:

- Let the system learn the complete trust logic.
- Addition of approximators (e.g. Neural Networks) that allow to improve and enhance the solution.
- Provide more powerful data analysis to support and improve the management of the files containing the different solutions.
- Implement a solution that includes two types of agents following a logic similar to the master-slave.
- Build a deep reinforcement learning solution and compare the results with our implementation.

Bibliography

[1] Alchemist official web site. https://alchemistsimulator.github.io/, accessed on: 2020-02-17.

[2] Scafi official web site. https://scafi.github.io/, accessed on: 2020-02-17.

[3] Alpaydin, E. *Introduction to Machine Learning*, 3 ed. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2014.

[4] Beal, J., and Viroli, M. Building blocks for aggregate programming of self-organising applications. In *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops* (Sep. 2014), pp. 8–13.

[5] Casadei, R., Aldini, A., and Viroli, M. Towards attack-resistant aggregate computing using trust mechanisms. *Science of Computer Programming 167* (2018), 114 – 137.

[6] Casadei, R., and Viroli, M. Towards aggregate programming in scala. In *PMLDC '16* (2016).

[7] Cetina, V. A multiagent architecture for concurrent reinforcement learning. pp. 107–112.

[8] COMARL. Challenges and opportunities for multi-agent reinforcement learning, 2019-2020. https://sites.google.com/view/comarl-aaai-2020/, accessed on: 2020-02-17.

[9] Das, P., Behera, H., and Panigrahi, B. Intelligent-based multi-robot path planning inspired by improved classical q-learning and improved particle swarm optimization with perturbed velocity. *Engineering Science and Technology, an International Journal 19*, 1 (2016), 651 – 669.

[10] Dowling, J., and Haridi, S. Decentralized reinforcement learning for the online optimization of distributed systems. Reinforcement Learning, Cornelius Weber, Mark Elshaw and Norbert Michael Mayer, IntechOpen.

[11] Guestrin, C., Lagoudakis, M. G., and Parr, R. Coordinated reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning* (San Francisco, CA, USA, 2002), ICML '02, Morgan Kaufmann Publishers Inc., pp. 227–234.

[12] Gunady, M. K., Gomaa, W., and Takeuchi, I. Multi-agent task division learning in hide-and-seek games. In *Artificial Intelligence: Methodology, Systems, and Applications* (Berlin, Heidelberg, 2012), A. Ramsay and G. Agre, Eds., Springer Berlin Heidelberg, pp. 256–265.

[13] J. Beal, D. P., and Viroli, M. Aggregate programming for the internet of things. *IEEE Computer magazine 48* (2015), 22–30.

[14] Kabysh, A., and Golovko, V. Collective behavior in multiagent systems based on reinforcement learning.

[15] Kwon, W. Y., Suh, I. H., Lee, S., and Cho, Y.-J. Fast reinforcement learning using stochastic shortest paths for a mobile robot. pp. 82 – 87.

[16] Lamini, C., Fathi, Y., and Benhlima, S. H-mas architecture and reinforcement learning method for autonomous robot path planning. In *2017 Intelligent Systems and Computer Vision (ISCV)* (April 2017), pp. 1–7.

[17] Leottau, D. L., del Solar, J. R., and Babuška, R. Decentralized reinforcement learning of robot behaviors. *Artificial Intelligence 256* (2018), 130 – 159.

[18] Mitchell, T. M. *Machine Learning*. McGraw-Hill, New York, 1997.

[19] Pianini, D., Montagna, S., and Viroli, M. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation 7*, 3 (2013), 202–215.

[20] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction*, second ed. The MIT Press, 2018.

[21] Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., and Pianini, D. From field-based coordination to aggregate computing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 10852 LNCS* (2018), 252–279. cited By 14.

[22] Viroli, M., Damiani, F., and Beal, J. A calculus of computational fields. In *Advances in Service-Oriented and Cloud Computing* (Berlin, Heidelberg, 2013), C. Canal and M. Villari, Eds., Springer Berlin Heidelberg, pp. 114–128.

[23] Zhang, C., Lesser, V., and Abdallah, S. Self-organization for coordinating decentralized reinforcement learning. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1* (Richland, SC, 2010), AAMAS '10, International Foundation for Autonomous Agents and Multiagent Systems, pp. 739–746.