Università degli Studi di Padova

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in
Ingegneria dell'Automazione

Consensus-based allocation of quotas in networked systems

Relatore:
Prof. Angelo Cenedese

Laureando:
Alvise Rossi
1134320

Anno Accademico 2020/2021
Data di laurea 13/12/2021

Abstract

Production processes whose output is represented by an additive quantity are countless. When there are multiple entities that must cooperatively produce a target quantity, the allocation of quotas of the target quantity to the single entities is often a task assigned to an external system, which acts as a coordinator. The first part of this thesis is focused on the definition of a distributed strategy for these processes: the entities capable of production are represented by agents of a Multi-Agent System, they can communicate over a network and are required to collectively produce a target amount of the related quantity, without being coordinated by an external system. The general framework described in the first part of this project is then applied using a Docker environment that allows to simulate the interaction among the agents on an actual network, so that the developed code is as close as possible to the one that would be deployed in a real-world scenario.

Contents

1 Introduction **1**

2 Problem Formulation **5**

3 Solutions **7**

3.1 The Exact-Breakdown method 8

3.1.1 Description of the method 8

3.1.2 The update rule and the state-space system 9

3.1.3 Characterization of matrices 10

3.1.4 Enforcing hard limits 13

3.2 The Self-Focused method 15

3.2.1 Description of the method 15

3.2.2 The update rule and the state-space system 18

3.2.3 Analysis of stability 20

3.2.4 Analysis of convergence time 23

3.2.5 Enforcing stability by design 28

3.2.6 Enforcing hard limits 30

3.2.7 Usage in MASs represented by sparse graphs 31

4 Implementation **35**

4.1 Usage of Docker Containers 36

4.2 Container specifications 39

4.2.1 Communication protocols 40

4.2.2 Programming languages and development techniques . . . 44

4.2.3 Implementation on sparse MASs 49

4.3 Simulation results 61

4.4 Activities to implement the code in a real-world scenario 67

5 Conclusions and further developments **69**

A Container code descriptions **73**

B List of software used in the project **85**

Acknowledgements **87**

References **89**

Chapter 1

Introduction

Many classes of consensus problems have obtained wide coverage in the literature. In [1], some of the most common are cited, including the Synchronization of Coupled Oscillators, Flocking Theory, Rendezvous in Space and Distributed Sensor Fusion in Sensor Networks. These classes of consensus problems are bound together by the fact that they focus on achieving an agreement among the agents estimates of a scalar variable, or a $\mathbb{R}^2$ or $\mathbb{R}^3$ vector: for the mentioned examples, the phase of the oscillators, the velocity of the agents, the position where to meet or the measured variable, respectively.

The class of problems studied in this work differs from the mentioned ones in that the variables to be agreed upon are as many as the number of agents involved. In fact, the aim of this work is to define a distributed framework for those processes in which a total quantity can be split into quotas to be assigned to a number of agents, allowing the agents themselves to agree on a partition that is accepted and applied by every agent. In [2], a scenario that could fall into this class of problems is presented, which is the one of dinamically load balancing the workload of a set of computing processors, but the goal of that process is to maintain an equal load for all the processors involved. Instead, with the framework proposed in this thesis, the goal is to let each agent propose its own partition basing upon an optimization computed locally, possibly with different criteria from agent to agent; then, perform a consensus process that finds the partition that represents the best trade-off among the various agents proposals. Thus, the final trade-off is not meant to assign the same quota to all the agents as in the load balancing case, but to assign to each agent a dedicated quota which fits best with all the proposals.

A problem that shares some common ground with the one presented in this

work is reported in [3]: in that case, the scenario is one where many individuals are required to express their opinion on a probability distribution for an unknown value of a parameter and an agreement must be found among all the opinion expressed. Although in this thesis the consensus problem is not related to a probability distribution, if the vectors representing the opinions on the probability distribution are replaced by vectors representing opinions on the deterministic partition of the total amount – with the main difference that the latter must have a dimension that matches the number of agents, the algebraic results still hold and the method proposed in [3] can be adapted to the problem addressed by this work. In fact, the solution to the problem presented in Subsection 3.1 is based on an autonomous system that, in algebraic terms, is identical to the one proposed in [3]. The wider range of application of the method beyond probability distributions was foreseen by the author, who mentioned the possibility to extend his theory to cases in which opinions *can be represented as points in some fixed convex set in an arbitrary linear space*. Also, throughout this thesis, the terms *opinion* and *weight* will be used with the same meaning with respect to [3].

The applications that could benefit from the framework described in this thesis are virtually unlimited. Some of these are related to the energy sector and, in fact, the idea for this work came from one of these: in the group project related to the *Networked Control for Multi-Agent Systems* course, a problem related to Secondary Control was addressed. The Secondary Control is a service provided by energy producers to maintain the stability of the grid, which description can be found in [4], and the author's group studied a procedure for a distributed allocation of quotas of the secondary control reserve that three hydroelectric plants were required to offer together. This thesis is meant to be an extension of that project, in the way that it generalizes the approach designed specifically for the three hydroelectric plants and it introduces other solutions and applications related to this problem.

Other applications in the energy sector could be related to microgrids, where the agreement could be sought by generating devices in relation to how much power each of them should supply to the microgrid, and to BESS systems, which have the particular aspect to be generally able to both produce and consume energy: this represents a particular application as the opinions involved would no longer be required to be non-negative variables. Moreover, a possible application on the consumption side is the distributed management of the *demand response* service, which is the controlled disconnection from the grid of home appliances

during peak load intervals associated with an economic reward granted to the appliance owner: when a particular area of the grid is required to reduce its consumption, the grid operator could just provide the consumers belonging to that area with the overall amount of load reduction requested to that area, then the group of consumers would agree on how much each of them will decrease the load, balancing their own economic interest with the urgency to keep their appliances running.

Such applications are likely to gain interest as the electrical grids move from the centralized and fossil fuel-based production paradigm that characterized the past decades to the distributed production model with increasing renewable resource participation expected for the following decades. This dramatic change means that a higher effort must be spent on the local side for maintaining the stability of the grid and, since this effort will involve a large number of producers and consumers, a distributed approach seems to be more robust and convenient as opposed to a centralized one, which could be less scalable than the distributed one and could make the fault management harder as distances between the center and the edge of the grid grow.

The thesis is structured as following: in Chapter 2 a formal definition of the general problem is presented, including a description of the specific terms that are used across the entire document. Chapter 3 is the main theoretical part where two possible solutions to the problem are presented, each of which applies best to a different set of contexts. In Chapter 4 the simulation of the consensus algorithms in a Docker environment is presented, with Appendix A providing the details of the developed code. Eventually, in Appendix B the software that was used for the study is listed.

Chapter 2

Problem Formulation

This study has the main purpose of providing general methods to solve problems that match the specifications reported in the following. Recurrent terms that will be used throughout the whole study are highlighted in bold.

- A Multi-Agent System (MAS) is composed by $n > 1$ agents that are able to communicate over some network. The agents represent control systems, each one with its own local inputs and outputs: the local inputs of a control system are not shared through the network with the other agents, nor the local outputs of a control system are directly influenced by other agents.

- The continuous time axis is divided into **time slots** of length $T \in \mathbb{R}, T > 0$ and the solution to the problem must be found for each time slot. The resulting discrete time intervals are identified by the variable $t \in \mathbb{N}$, which is relative to a generic continuous time instant $T_0 \in \mathbb{R}$. This means that the t-th time interval corresponds to the time range $[T_0 + t \cdot T, T_0 + (t+1) \cdot T]$.

- For each time interval t, a signal $\bar{u}(t) \in \mathbb{R}$ representing the **total amount** of some quantity is provided by an external system to all the agents of the MAS. The total amount $\bar{u}(t)$ must be divided into n **quotas**, each one associated with the n agents forming the network. Depending on the specific application, the total amount and the quotas can be defined as non-negative variables. The process of defining the fraction of the total amount that is assigned to each agent is referred to as **allocation**. This process leads to the definition of quotas where the i-th quota is referred to as $x_i^{(\infty)}(t), 1 \leq i \leq n$. Trivially, quotas are constrained by the fact that they must collectively

cover the whole total amount $\bar{u}(t)$, that is:

$$\sum_{i=1}^{n} x_i^{(\infty)}(t) = \bar{u}(t) \tag{2.1}$$

The vector stacking all the allocated quotas $x^{(\infty)}(t) = \begin{bmatrix} x_1^{(\infty)}(t) & x_2^{(\infty)}(t) & \cdots & x_n^{(\infty)}(t) \end{bmatrix}^T$ is called the **breakdown** of the total amount corresponding to the t-th time slot.

- The allocation process must follow a distributed approach, where each agent must have a way to influence the quota it will be assigned by the allocation process basing upon some optimization process computed locally. The optimization processes must be performed on the basis of the local inputs and outputs of the specific agent. The breakdown is then determined as an agreement among the results of all the optimization processes of the various agents.

- The allocation process must be designed for providing a result within a predefined amount of time $\tau^{MAX} \in \mathbb{R}$, where $\tau^{MAX} \leq T$. Extraordinary cases in which the allocation process is not able to provide a result within the time limit must be handled by a dedicated fallback logic.

- Agents must also have a way to enforce **hard limits**, which are boundaries on the quota they can be assigned during the time interval t. They are defined by the variables $x_i^{MIN}(t)$ and $x_i^{MAX}(t)$ and introduce a new constraint formally expressed by:

$$x_i^{MIN}(t) \leq x_i^{(\infty)}(t) \leq x_i^{MAX} \tag{2.2}$$

Moreover, due to the linear nature of the problem, the set of hard limits imposed by all the agents introduces a constraint that must be respected by the total amount $\bar{u}(t)$ for the allocation process to be successful:

$$\sum_{i=1}^{n} x_i^{MIN}(t) \leq \bar{u}(t) \leq \sum_{i=1}^{n} x_i^{MAX}(t) \tag{2.3}$$

Chapter 3

Solutions

Before exploring the two solutions that were designed to solve the problem, some common concepts and differences are expressed.

Both the presented solutions are implemented through an iterative approach based on the concept of **opinion**, which represents a candidate quota to be assigned to one agent – they are therefore defined on the same space of their associated quotas $x_i^{(\infty)}(t), 1 \leq i \leq n$. Throughout the allocation process, agents make their opinions evolve considering the opinions received by other agents, until all the opinions are consistent with the constraints imposed by the problem and with each other.

The main difference between the two methods is described in the following:

- In the *Exact-Breakdown* method each agent holds opinions about the candidate quota for all the agents, including itself. This leads to a number of n^2 opinions defined by $x_{ij}^{(k)}(t), i, j \in [1, n], k \in \mathbb{N}$, representing the opinion computed by agent i on the candidate quota to be assigned to agent j at the iteration k of the time slot t.

- In the *Self-Focused* method every agent holds an opinion about the candidate quota to be assigned to itself only, which is $x_i^{(k)}(t), 1 \leq i \leq n, k \in \mathbb{N}$.

With regards to time slot t, each agent i initially sets its **initial opinion** $x_i^{(0)}(t)$ (Self-Focused method) or set of opinions $x_{ij}^{(0)}(t)$ (Exact-Breakdown method) according to the outcome of the optimization process that was executed locally. Basically, initial opinions are the representation of how each agent aims to allocate the quotas of the total amount to itself and to the other agents (collectively in the Self-Focused case).

With the initial opinion, agents express their preferred candidate quotas. The allocation process, while seeking an agreement among all the initial opinions, may provide results that are largely different from an agent preference: for some agents this might not represent a critical event, while for some others it could lead to relevant drawbacks. In order to provide a tool for agents to express their level of flexibility with respect to the initial opinion they provide, **weights** are introduced. With reference to the time slot t, weights are defined as:

- $a_{ij}(t) \in \mathbb{R}, 0 \leq a_{ij} \leq 1, 1 \leq i \leq n, 1 \leq j \leq n$, with the meaning of *the weight assigned from agent i to the opinion of agent j*, in case the Exact-Breakdown method is used
- $a_i(t) \in \mathbb{R}, 0 \leq a_i \leq 1, 1 \leq i \leq n$, with the meaning of *the weight agent i assigns to its opinion*, in case the Self-Focused method is used

After the definition of the initial opinion and weights by all the agents, the agreement process start, consisting of an **update rule** that is iterated. The update rule is different in the two methods and is explained in detail in the related section. As in many cases the iteration of the update rule leads to an asymptotic behaviour for the opinions, a stop condition must be defined for both methods when the changes provided by the application of the update rule are no longer relevant, then the breakdown $x^{(\infty)}(t)$ is derived by the value assumed by the opinions at the last iteration.

3.1 The Exact-Breakdown method

3.1.1 Description of the method

The Exact-Breakdown method owes its name to the fact that every agent holds opinions about the candidate quota for all the agents of the network, which means that at the end of the allocation process every agent holds *an exact copy of the breakdown* $x^{(\infty)}(t)$. This is the root difference with respect to the method exposed in the following section, in which an agent does not hold a specific opinion for each of the other agents, thus it is never aware of the breakdown of the total amount.

It is also the method that is more similar to traditional consensus approaches, as it is the natural extension of the consensus problem for a scalar variable to the vector space. Indeed, at least for what concerns the normal situation when no

agent is hitting a hard limit, this solution resembles the aggregation of n separate scalar consensus problems, each one related to the quota to be assigned to one of the agents. As was mentioned in the introduction, the algebraic structure of this method is closely related to the one reported in [3], which was applied for finding an agreement among actual opinions expressed by individuals.

3.1.2 The update rule and the state-space system

The update rule of the Exact-Breakdown method is essentially based on a weighted average, referred to as *Linear Opinion Pool* in [3]: every agent updates its opinion on the candidate quota to be assigned to a specific agent – itself included – weighting the current opinion of all the agents on the same candidate quota. Considering the update of the opinion of agent i on the quota to be assigned to agent j during time slot t, the update rule is formally expressed by the following equation:

$$x_{ij}^{(k+1)}(t) = a_{i1}(t)x_{1j}^{(k)}(t) + a_{i2}(t)x_{2j}^{(k)}(t) + \cdots + a_{in}(t)x_{nj}^{(k)}(t) = \sum_{l=1}^{n} a_{il}(t)x_{lj}^{(k)}(t) \tag{3.1}$$

In order to study the overall evolution of the opinions and the constraints to be imposed to meet the specifications, the n^2 update rules associated with the evolution of each agent's opinion related to each agent's candidate quota are grouped in the following matrix equation:

$$\begin{bmatrix} x_{11}^{(k+1)}(t) & x_{12}^{(k+1)}(t) & \cdots & x_{1n}^{(k+1)}(t) \\ x_{21}^{(k+1)}(t) & x_{22}^{(k+1)}(t) & \cdots & x_{2n}^{(k+1)}(t) \\ \vdots & \ddots & \ddots & \vdots \\ x_{n1}^{(k+1)}(t) & x_{n2}^{(k+1)}(t) & \cdots & x_{nn}^{(k+1)}(t) \end{bmatrix} = \begin{bmatrix} a_{11}(t) & a_{12}(t) & \cdots & a_{1n}(t) \\ a_{21}(t) & a_{22}(t) & \cdots & a_{2n}(t) \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1}(t) & a_{n2}(t) & \cdots & a_{nn}(t) \end{bmatrix} \begin{bmatrix} x_{11}^{(k)}(t) & x_{12}^{(k)}(t) & \cdots & x_{1n}^{(k)}(t) \\ x_{21}^{(k)}(t) & x_{22}^{(k)}(t) & \cdots & x_{2n}^{(k)}(t) \\ \vdots & \ddots & \ddots & \vdots \\ x_{n1}^{(k)}(t) & x_{n2}^{(k)}(t) & \cdots & x_{nn}^{(k)}(t) \end{bmatrix}$$

$$X^{(k+1)}(t) = A(t)X^{(k)}(t) \tag{3.2}$$

In Eq. 3.2, matrix $X^{(k)}(t)$ groups all the opinions of the agents at iteration k. By extracting a row, one obtains all the opinions of a specific agent on how the total amount should be allocated among the agents; by extracting a column, one obtains all the opinions of the various agents on the quota to be assigned to a specific agent. In fact, the similarity with the scalar consensus problem comes from the fact that column j of the matrix $X(t)$ updates through the equations

represented by

$$\begin{bmatrix} x_{1j}^{(k+1)}(t) \\ x_{2j}^{(k+1)}(t) \\ \vdots \\ x_{nj}^{(k+1)}(t) \end{bmatrix} = \begin{bmatrix} a_{11}(t) & a_{12}(t) & \cdots & a_{1n}(t) \\ a_{21}(t) & a_{22}(t) & \cdots & a_{2n}(t) \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1}(t) & a_{n2}(t) & \cdots & a_{nn}(t) \end{bmatrix} \begin{bmatrix} x_{1j}^{(k)}(t) \\ x_{2j}^{(k)}(t) \\ \vdots \\ x_{nj}^{(k)}(t) \end{bmatrix} \tag{3.3}$$

which represents the scalar consensus problem on the quota to be assigned to agent j: the expected behaviour is that the opinion of all agents on the quota to be assigned to agent j converge to a common value, that is

$$x_{1j}^{(\infty)}(t) = x_{2j}^{(\infty)}(t) = \cdots x_{nj}^{(\infty)}(t) \implies \begin{bmatrix} x_{1j}^{(\infty)}(t) \\ x_{2j}^{(\infty)}(t) \\ \vdots \\ x_{nj}^{(\infty)}(t) \end{bmatrix} = x_j^{(\infty)}(t)\mathbb{1}$$

where $x_j^{(\infty)}$ represents the quota eventually assigned to agent j, which is the j-th component of the breakdown vector $x^{(\infty)}(t)$.

3.1.3 Characterization of matrices

For the sake of readability, the dependence on the time slot t is dropped throughout this Section, as the results of the matrices characterization apply to every time slot independently.

As the problem definition states that all the agents of the network receive the value of the total amount $\bar{u}$ from the external system and each agent is required to compute its initial set of opinions ($x_{ij}^{(0)}, 1 \leq j \leq n$ for agent i), it is easy to also require that the sum of the initial opinions of an agent is equal to the total amount. This can be enforced through the optimization logic deployed to the local control systems that represent the agents of the network.

In the Exact-Breakdown method, the way the constraint on the sum of the breakdown components given by Eq. 2.1 is implemented is by ensuring that each iteration of the update rule does not change the sum of the opinions held by an agent. This imples that, if agent i computes an initial set of opinions $x_{ij}^{(0)}, 1 \leq j \leq n$ that covers the total amount $\bar{u}$, $\bar{u}$ will correspond also to the sum of its opinions at the end of the allocation process, i.e. $x_{ij}^{(\infty)}$.

Proposition 3.1.1. *The necessary and sufficient condition for the update rule of*

Eq. 3.2 to keep the same value for the sum of opinions of an agent is that $\boldsymbol{A}$ *is row-stochastic.*

Proof. The constraint imposing that the sum of the opinions of an agent is equal to the total amount, using the matrix notation defined in Subsection 3.1.2, requires that the sum by rows of the matrix $X^{(k)}$ is equal to the vector $\begin{bmatrix}\bar{u} & \bar{u} & \cdots & \bar{u}\end{bmatrix}^T$, that is:

$$X^{(k)} \cdot \mathbb{1} = \begin{bmatrix} x_{11}^{(k)} & x_{12}^{(k)} & \cdots & x_{1n}^{(k)} \\ x_{21}^{(k)} & x_{22}^{(k)} & \cdots & x_{2n}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1}^{(k)} & x_{n2}^{(k)} & \cdots & x_{nn}^{(k)} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} \bar{u} \\ \bar{u} \\ \vdots \\ \bar{u} \end{bmatrix} = \bar{u} \cdot \mathbb{1} \tag{3.4}$$

The proposition is proven by induction: the base case $X^0 \cdot \mathbb{1} = \bar{u} \cdot \mathbb{1}$ is verified by construction as the local optimization process executed by the agents is required to provide consistent initial opinions. Then, assuming that the (3.4) holds for a certain iteration k and considering how the i-th row of the state matrix updates, we have the following:

$$[x_{i1}^{(k+1)} \quad x_{i2}^{(k+1)} \quad \cdots \quad x_{in}^{(k+1)}] = [a_{i1} \quad a_{i2} \quad \cdots \quad a_{in}] \cdot X^{(k)} \tag{3.5}$$

Right-multiplying both sides by $\mathbb{1}$ we have:

$$\begin{aligned} &[x_{i1}^{(k+1)} \quad x_{i2}^{(k+1)} \quad \cdots \quad x_{in}^{(k+1)}] \cdot \mathbb{1} = [a_{i1} \quad a_{i2} \quad \cdots \quad a_{in}] \cdot X^{(k)} \cdot \mathbb{1} \\ &= \begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{in} \end{bmatrix} \cdot \bar{u} \cdot \mathbb{1} = \begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{in} \end{bmatrix} \cdot \mathbb{1} \cdot \bar{u} \end{aligned} \tag{3.6}$$

where the inductive hypothesis $X^{(k)} \cdot \mathbb{1} = \bar{u} \cdot \mathbb{1}$ was used.

The sum of the i-th row of the matrix is $\bar{u}$ at the iteration $k+1$ when

$$\begin{bmatrix} x_{i1}^{(k+1)} & x_{i2}^{(k+1)} & \cdots & x_{in}^{(k+1)} \end{bmatrix} \cdot \mathbb{1} = \bar{u} \tag{3.7}$$

which, from (3.6), becomes

$$\begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{in} \end{bmatrix} \cdot \mathbb{1} \cdot \bar{u} = \bar{u} \tag{3.8}$$

that, for $\bar{u} \neq 0$, is true if and only if it holds that

$$\begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{in} \end{bmatrix} \cdot \mathbb{1} = 1 \tag{3.9}$$

This means that the necessary and sufficient condition to guarantee that the agreement process preserves the value of the sum of the opinions of agent i is

that:

$$\sum_{l=1}^{n} a_{il} = 1 \tag{3.10}$$

which means that the i-th row of matrix A must sum up to 1. The same reasoning can be applied to the opinions of all the other agents, resulting in the requirement that all the rows of matrix A must sum up to 1, i.e. matrix A must be a row-stochastic matrix. □

A similar statement and proof in the context of computational load balancing can be found in Lemma 1 of [2], where the focus is on the average of the workloads assigned to processors: it is not a relevant difference as the sum of workloads equals to the average times the fixed number of agents, therefore if the average does not change through iterations, neither does the sum of the workloads.

The following proposition sets the conditions for the Exact-Breakdown method to be successful, that is, for the sets of agents' opinions to converge to the same vector – the breakdown.

Proposition 3.1.2. *A sufficient condition for the agent opinions to converge to an equal vector is that A is a row-stochastic and positive matrix.*

Proof. For this analysis of convergence, we will consider how the columns of the matrix $X^{(k)}$ update at each iteration. According to Eq. 3.3, the update of each column of matrix $X^{(k)}$ resembles the usual scalar consensus problem: in this context, a corollary of the Perron-Frobenius theorem states that, if matrix A is row-stochastic and primitive, the states of the consensus problem converge in a linear combination of the initial conditions. Applied to our case, the corollary means that

$$\lim_{k\to\infty} A^k \begin{bmatrix} x_{1i}^0 \\ x_{2i}^0 \\ \vdots \\ x_{ni}^0 \end{bmatrix} = \alpha \mathbb{1} = w_0^T \begin{bmatrix} x_{1i}^0 \\ x_{2i}^0 \\ \vdots \\ x_{ni}^0 \end{bmatrix}$$

with w_0 eigenvector associated with the largest eigenvalue of A, that is $\lambda_0 = 1$.

The fact that matrix A is row-stochastic is already required by Proposition 3.1.1. On the other hand, ensuring that the matrix is generically primitive is something that could not be enforced by single agents without exchanging the information about each other's weight. Instead, requiring the matrix to be positive is a constraint that each agent can trivially enforce independently from each other; as the positivity of a matrix is a stricter condition that implies primitivity,

it is a sufficient condition for consensus. In practical terms, in order to build a positive matrix A, it is required that every agent assigns a weight greater than zero to every agents' opinions, that is

$$a_{ij} > 0, \ \forall i, j$$

□

Corollary 3.1.2.1. *A sufficient condition for ensuring both the convergence of opinions and the compliance with constraint 2.1 is that every agent assign weights such that*

$$0 < a_{ij} < 1, \forall i, j$$

and that they sum up to 1*, that is* $\sum_{j=1}^{n} a_{ij}(t) = 1, \forall i$.

Proof. Directly from Proposition 3.1.2: since $n > 1$, for the weights assigned by an agent to be both positive and have unitary sum they must also be lower than one. □

3.1.4 Enforcing hard limits

The system studied so far does not handle situations in which agents are bounded by some hard limits on the minimum or maximum quota they can take care of, which was a requirement of the problem defined in Chapter 2. Recall that hard limits mean that during the time slot t the agent i is bounded by

$$x_i^{MIN}(t) \leq x_i^{(\infty)}(t) \leq x_i^{MAX}(t)$$

A simple example is again the context of power generation, where the plants represented by the agents cannot generate a negative amount of energy — in some cases they cannot even decrease the power below a minimum positive threshold — and are capped by a maximum limit.

The solution that was applied to enforce hard limits was to introduce an input $\widetilde{U}^{(k)}(t)$ to the autonomous system defined by Eq. 3.2. The update rule would then become

$$X^{(k+1)}(t) = A(t)X^{(k)}(t) + \widetilde{U}^{(k)}(t) \tag{3.11}$$

In the proposed approach, the value of the input is computed by agent i only in relation to its own hard limits $x_i^{MIN}(t)$ and $x_i^{MAX}(t)$ and is designed in such a way that it counteracts the action of the autonomous update rule of Eq. 3.2 when

the latter would bring its opinion about its quota $x_{ii}^{(k)}(t)$ outside the acceptable range, keeping it at the limit value instead. Using the Heaviside step function, defined as

$$\theta(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

the counteraction input of agent i is represented by

$$\begin{aligned} u_i^{(k)}(t) = &- \theta\left(\sum_{l=1}^{n} a_{il}(t)x_{li}^{(k)}(t) - x_i^{MAX}(t)\right) \cdot \left(\sum_{l=1}^{n} a_{il}(t)x_{li}^{(k)}(t) - x_i^{MAX}(t)\right) + \\ &+ \theta\left(x_i^{MIN}(t) - \sum_{l=1}^{n} a_{il}(t)x_{li}^{(k)}(t)\right) \cdot \left(x_i^{MIN}(t) - \sum_{l=1}^{n} a_{il}(t)x_{li}^{(k)}(t)\right) = \\ = &- u_{i+}^{(k)}(t) + u_{i-}^{(k)}(t) \end{aligned} \tag{3.12}$$

where the first term accounts for the counteraction of excesses of the updated opinion $x_{ii}^{(k+1)}(t)$ with respect to $x_i^{MAX}(t)$ and the second accounts for the conteraction of defects with respect to $x_i^{MIN}(t)$.

In order to comply with the basic constraint of maintaining a set of opinions that is consistent with the total amount $\bar{u}(t)$ – which must be enforced at each iteration of the update rule for the breakdown $x^{(\infty)}(t)$ to sum up to $\bar{u}(t)$ – the row sum of the opinion matrix $X^{(k+1)}(t)$ must not be changed by the introduction of the input with respect to the row sum $X^{(k)}(t)$. This can be accomplished by imposing that the row sum of matrix $\widetilde{U}^{(k)}(t)$ of Eq. 3.11 is zero, as shown by the following chain of equations:

$$X^{(k+1)}(t) \cdot \mathbb{1} = \left(A(t)X^{(k)}(t) + \widetilde{U}^{(k)}(t)\right) \cdot \mathbb{1} = A(t)X^{(k)}(t) \cdot \mathbb{1} + \widetilde{U}^{(k)}(t) \cdot \mathbb{1} = A(t)X^{(k)}(t) \cdot \mathbb{1}$$

Imposing that the rows of $\widetilde{U}^{(k)}(t)$ have zero sum means to require that agents compensate the input they apply to their opinion about their quota (Eq. 3.12) with an opposite change in the sum of the opinions about the quota to be assigned to other agents. This can be achieved by breaking down the input $\widetilde{U}^{(k)}(t)$ as the

product of two matrices:

$$\begin{aligned}\widetilde{U}^{(k)}(t) &= U^{(k)}(t) \cdot B = \\ &= \begin{bmatrix} u_1^{(k)}(t) & 0 & \cdots & 0 \\ 0 & u_2^{(k)}(t) & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & u_n^{(k)}(t) \end{bmatrix} \begin{bmatrix} 1 & b_{12} & \cdots & b_{1n} \\ b_{21} & 1 & \cdots & b_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & 1 \end{bmatrix} = \\ &= \begin{bmatrix} u_1^{(k)}(t) & b_{12}u_1^{(k)}(t) & \cdots & b_{1n}u_1^{(k)}(t) \\ b_{21}u_2^{(k)}(t) & u_2^{(k)}(t) & \cdots & b_{2n}u_2^{(k)}(t) \\ \vdots & \ddots & \ddots & \vdots \\ b_{n1}u_n^{(k)}(t) & b_{n2}u_n^{(k)}(t) & \cdots & u_n^{(k)}(t) \end{bmatrix}\end{aligned} \tag{3.13}$$

With this notation, the constraint on the row sum of the input matrix can be enforced by making the rows of matrix B sum up to zero, that is to impose $\sum_{i \neq j} b_{ij} = -1$. In this context, coefficients b_{ij} represent parameters whose purpose is to define how much agent i relies on agent j for balancing its counteraction $u_i^{(k)}(t)$.

As long as the constraint on the range of the total amount $\bar{u}(t)$ given by Eq. 2.3 is satisfied, it is trivial to recognise that not all the agents will activate their input together, therefore the convergence to a common breakdown vector is ensured. Nevertheless, the introduction of the input slows down the allocation process, with consequences on the convergence time that must be investigated further.

3.2 The Self-Focused method

3.2.1 Description of the method

The main drawback of the Exact-Breakdown method described in the previous section is that every agent involved, besides expressing the quota that aims to assign to itself, must also express an opinion about the specific quota to be assigned to each of the other agents. This can be useful when agents are aware of the dynamics underlying their mutual relationship at the physical level, and also share the state of each time slot: for instance, if a hydropower plant wants to increase its production, it can also try to make the upstream plant increase its production so that it will receive a higher water flow.

On the other hand, this method is not really suitable for scenarios where the number of agents involved is large, as using that method the overall number of opinions to be computed and transmitted across the network increases with the square of the number of agents. Since one of the advantages of the decentralized solution to the problem addressed in this study is that it can be deployed to infrastructures where the computing power and the network bandwidth are low, a different solution must be found for environments with a large number of agents.

Moreover, using the Exact-Breakdown method in scenarios where the agents are not bound by some common physical dynamics, is also likely to result in suboptimal allocation of quotas. This is due to the fact that an agent is always required to express an opinion about the quota to be assigned to every other agent, even when it has no a-priori information about the mutual physical relation with them and from its standpoint the exact breakdown of quotas among the other agents is not relevant at all. In that case, every agent might express opinions about other agents without any meaningful procedure, for instance by dividing equally among the other agents the remaining part of the total, after defining its opinion about itself. This has an influence on the final allocation that should be avoided in order to obtain the best allocation possible for all agents. Also, the structure of the Exact-Breakdown procedure does not allow to weight differently the various opinions expressed by the same agent, so if an agent aims to have a lower influence on the allocation related to other agents it is forced to have less influence also on the quota eventually assigned to itself.

Referring to the example of hydropower plants production, assume that three of them are not placed along the same river, they are instead totally independent to one another in terms of water stream involved, and they need to cooperatively generate an active power of 90MW. As a limit example, assume that they all weight every opinion equally, that is $a_{ij} = \frac{1}{3}, \forall i, j$ (more generally, nothing changes in what follows if the self-weights a_{ii} are equal to each other and so do the remaining weights $a_{ij}, i \neq j$), but the first of the three aims to take care of the whole amount of 90MW and the other two aim to not generate power at all. The most reasonable allocation of quotas would be to actually assign 90MW to the first agent and leave the other two inactive but, if the second and third agents assign the quotas related to the other agents by dividing the remaining power equally, their opinion will be that other agents must generate 45MW each, that

is:

$$X^{(0)}(t) = \begin{bmatrix} x_{11}^{(0)}(t) & x_{12}^{(0)}(t) & x_{13}^{(0)}(t) \\ x_{21}^{(0)}(t) & x_{22}^{(0)}(t) & x_{23}^{(0)}(t) \\ x_{31}^{(0)}(t) & x_{32}^{(0)}(t) & x_{33}^{(0)}(t) \end{bmatrix} = \begin{bmatrix} 90 & 0 & 0 \\ 45 & 0 & 45 \\ 45 & 45 & 0 \end{bmatrix}$$

The breakdown of power setpoint in these conditions would be $\left[60,\ 15,\ 15\right]$, bringing all the three agents away from their preferred condition without any significant benefit. Different approaches that might allocate the opinions on other agents' quotas through more thoughtful procedures are expected to mitigate the issue, but they do not solve it completely. A completely different approach has been studied to overcome the mentioned issues, which will be referred to as *Self-Focused* method. This method requires that, for every time slot t:

- As for the Exact-Breakdown method, every agent is initially informed by an external system of the total amount $\bar{u}$ to be divided among all the agents. Also, all agents have the capability of exchanging information with all the other agents, although this hypothesis will be relaxed in Section 3.2.7;
- Every agent holds an opinion about the quota to be assigned to itself only, which is called $x_i^{(k)}(t)$; implicitly, an opinion $\widetilde{x}_i^{(k)}(t)$ about the complementary quota to be assigned collectively to the other agents is defined as the difference between the total amount $\bar{u}(t)$ and $x_i^{(k)}(t)$, i.e. $\widetilde{x}_i^{(k)}(t) = \bar{u}(t) - x_i^{(k)}(t)$;
- Similarly, every agent chooses a single weight assigned to itself, which is called $a_i(t)$; also in this case, an implicit weight $\widetilde{a}_i(t)$ that represents the weight associated with the opinion of other agents is defined as $\widetilde{a}_i(t) = 1 - a_i(t)$.

This procedure is based on a weighted average of $\widetilde{x}_i^{(k)}(t)$ and $\sum_{j \neq i} x_j^{(k)}(t)$: also in this case every agent updates its opinion through the method called *linear opinion pool* in [3], but in this case the linear combination involves only two terms:

- the one related to the amount the agent aims not to take care of, and
- the sum of the candidate quotas that the other agents aim to assign themselves.

As will be shown in what follows, if some conditions are met – and they can be enforced by design, the sum of opinions $\sum_{i=1}^{n} x_i^{(k)}(t)$ for $k \to \infty$ converges to the total amount $\bar{u}(t)$ requested to the group of agents. One main difference with the Exact-Breakdown method is that, in the general case, the constraint about

having opinions consistent with the total amount is reached asymptotically and cannot be enforced at each step of the consensus; in the Exact-Breakdown method instead, at each iteration every agent held a valid n-uple of opinions, although different agents held different n-uples until convergence happened, making this difference not so relevant.

3.2.2 The update rule and the state-space system

For what concerns the opinion held by a single agent, the update rule is derived from the following equation:

$$\widetilde{x}_i^{(k+1)}(t) = a_i(t)\widetilde{x}_i^{(k)}(t) + \widetilde{a}_i(t) \sum_{j \neq i} x_j^{(k)}(t)$$

which states that the updated opinion of agent i about its complementary quota is a linear combination of two terms:

- the current value of the same quantity;
- the sum of the other agents' opinion about their own quotas

Intuitively, for the update rule to bring to an agreement, this linear combination must represent a weighted average, that means $0 \leq a_i(t) \leq 1$; this requirement will also be needed for converging to a set of states $x^{(\infty)}(t)$ that sums up to $\bar{u}(t)$, as will be proven later on.

By recalling that $\widetilde{x}_i^{(k)}(t) = \bar{u}(t) - x_i^{(k)}(t)$ and that $\widetilde{a}_i(t) = 1 - a_i(t)$, the equation can be rewritten as:

$$\bar{u}(t) - x_i^{(k+1)}(t) = a_i(t)\left(\bar{u} - x_i^{(k)}(t)\right) + (1 - a_i(t)) \sum_{j \neq i} x_j^{(k)}(t)$$

$$x_i^{(k+1)}(t) = a_i(t)x_i^{(k)}(t) + (a_i(t) - 1) \sum_{j \neq i} x_j^{(k)}(t) + (1 - a_i(t))\,\bar{u}(t) \tag{3.14}$$

With the purpose of studying the evolution of the system made of the whole set of agents, the equations of the single agents can be stacked, achieving the following

state-space system:

$$x^{(k+1)}(t) = \begin{bmatrix} x_1^{(k+1)}(t) \\ x_2^{(k+1)}(t) \\ \vdots \\ x_n^{(k+1)}(t) \end{bmatrix} = \begin{bmatrix} a_1 & a_1 - 1 & \cdots & a_1 - 1 \\ a_2 - 1 & a_2 & \cdots & a_2 - 1 \\ \vdots & \ddots & \ddots & \vdots \\ a_n - 1 & a_n - 1 & \cdots & a_n \end{bmatrix} \begin{bmatrix} x_1^{(k)}(t) \\ x_2^{(k)}(t) \\ \vdots \\ x_n^{(k)}(t) \end{bmatrix} + \begin{bmatrix} 1 - a_1 \\ 1 - a_2 \\ \vdots \\ 1 - a_n \end{bmatrix} \bar{u} = \tag{3.15}$$
$$= Ax^{(k)}(t) + B\bar{u}(t)$$

Proposition 3.2.1. *When the system asymptotically converges, it reaches a set of values that actually meets the required condition of covering the total demand $\bar{u}(t)$, that is*

$$\lim_{k\to\infty} \sum_{i=1}^{n} x_i^{(k)}(t) = \bar{u}(t)$$

Proof. For the sake of easiness, during this proof the dependence on the time slot t will be dropped. If the system asymptotically converges, it holds that $\lim_{k\to\infty} \left(x^{(k+1)} - x^{(k)}\right) = 0$, therefore at steady state the update equation can be rewritten as

$$x^{(k)} = Ax^{(k)} + B\bar{u}$$

$$(I - A)\, x^{(k)} = B\bar{u}$$

$$\begin{bmatrix} 1 - a_1 & 1 - a_1 & \cdots & 1 - a_1 \\ 1 - a_2 & 1 - a_2 & \cdots & 1 - a_n \\ \vdots & \ddots & \ddots & \vdots \\ 1 - a_n & 1 - a_n & \cdots & 1 - a_n \end{bmatrix} \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{bmatrix} = \begin{bmatrix} 1 - a_1 \\ 1 - a_2 \\ \vdots \\ 1 - a_n \end{bmatrix} \bar{u}$$

$$\begin{bmatrix} 1 - a_1 \\ 1 - a_2 \\ \vdots \\ 1 - a_n \end{bmatrix} [1 \quad 1 \quad \cdots \quad 1] \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{bmatrix} = \begin{bmatrix} 1 - a_1 \\ 1 - a_2 \\ \vdots \\ 1 - a_n \end{bmatrix} \bar{u}$$

$$\begin{bmatrix} 1 - a_1 \\ 1 - a_2 \\ \vdots \\ 1 - a_n \end{bmatrix} \sum_i x_i^{(k)} = \begin{bmatrix} 1 - a_1 \\ 1 - a_2 \\ \vdots \\ 1 - a_n \end{bmatrix} \bar{u}$$

$$\sum_i x_i^{(k)} = \bar{u}$$

□

3.2.3 Analysis of stability

In order to study the stability of the system, which ensures that all the opinions asymptotically reach a constant value, the eigenvalues and eigenvectors of matrix A are analysed. The dependence on the specific time slot t is disregarded also for this section.

Proposition 3.2.2. *Regardless of the value of n, the structure of matrix A yields to:*

1. *A set of eigenvalues* $\lambda_1 = \lambda_2 = \ldots = \lambda_{n-1} = 1$*, with multiplicity (both algebraic and geometrical) equal to* $n-1$ *and corresponding eigenvectors* $v_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ -1 \end{bmatrix}$, $v_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ -1 \end{bmatrix}$, $\cdots$, $v_{n-1} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ -1 \end{bmatrix}$;
2. *A single eigenvalue* $\lambda_n = \sum_i a_i - (n-1)$*, with corresponding eigenvector equal to* $v_n = \begin{bmatrix} a_1 - 1 \\ a_2 - 1 \\ \vdots \\ a_{n-1} - 1 \\ a_n - 1 \end{bmatrix}$.

The proof of the statement is reported below.

Proof. 1. First of all, the proof that $\lambda = 1$ is an eigenvalue of the matrix A is directly shown by the fact that the matrix

$$A - \lambda I = A - I = \begin{bmatrix} a_1 - 1 & a_1 - 1 & \cdots & a_1 - 1 \\ a_2 - 1 & a_2 - 1 & \cdots & a_2 - 1 \\ \vdots & \ddots & \ddots & \vdots \\ a_n - 1 & a_n - 1 & \cdots & a_n - 1 \end{bmatrix}$$

is trivially rank-deficient as all the columns are equal, therefore its determinant must be zero, meaning that 1 is an eigenvalue for A.

The proof that this eigenvalue has geometrical multiplicity equal to $n-1$ is directly shown:

$$Av = \lambda v$$

$$(A - 1I)\, v = 0$$

$$\begin{bmatrix} a_1 - 1 & a_1 - 1 & \cdots & a_1 - 1 \\ a_2 - 1 & a_2 - 1 & \cdots & a_2 - 1 \\ \vdots & \ddots & \ddots & \vdots \\ a_n - 1 & a_n - 1 & \cdots & a_n - 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = 0$$

$$\begin{cases} (a_1 - 1)\,(v_1 + v_2 + \cdots + v_n) = 0 \\ (a_2 - 1)\,(v_1 + v_2 + \cdots + v_n) = 0 \\ \vdots \\ (a_n - 1)\,(v_1 + v_2 + \cdots + v_n) = 0 \end{cases} \tag{3.16}$$

Which, in the general case with $a_i \neq 1 \ \forall i$ (which will later be required to be enforced for stability), yields to the single constraint expressed by

$$v_1 + v_2 + \cdots + v_n = 0 \Rightarrow v_n = -v_1 - v_2 - \cdots - v_{n-1}$$

The associated eigenspace is then defined by the eigenvectors

$$< \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ -1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ -1 \end{bmatrix}, \ \ldots, \ \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ -1 \end{bmatrix} >$$

which dimension proves that the eigenvalue $\lambda = 1$ has geometrical multiplicity equal to $n-1$. The algebraic multiplicity of the eigenvalue must therefore be greater or equal than $n - 1$. In order to prove that also the algebraic multiplicity is exactly $n - 1$, it is sufficient to find a different eigenvalue to reach a number of single eigenvalues equal to n, and this will be shown in the following point.

2. The fact that $\lambda_n = \sum_i a_i - (n-1)$ is an eigenvalue of A is also shown by direct proof. Matrix $A - (\sum_i a_i - (n-1))\, I$ is equal to

$$\begin{bmatrix} -\sum_{i \neq 1} (a_i - 1) & a_1 - 1 & \cdots & a_1 - 1 \\ a_2 - 1 & -\sum_{i \neq 2} (a_i - 1) & \cdots & a_2 - 1 \\ \vdots & \ddots & \ddots & \vdots \\ a_n - 1 & a_n - 1 & \cdots & -\sum_{i \neq n} (a_i - 1) \end{bmatrix}$$

For the sake of clarity, the matrix is rewritten by using the different set of parameters $b_i = a_i - 1$:

$$\begin{bmatrix} -\sum_{i\neq 1} b_i & b_1 & \cdots & b_1 \\ b_2 & -\sum_{i\neq 2} b_i & \cdots & b_2 \\ \vdots & \ddots & \ddots & \vdots \\ b_n & b_n & \cdots & -\sum_{i\neq n} b_i \end{bmatrix}$$

If we study the linear independence of the rows of the matrix using coefficients k_j, $j = 1, ..., n$, the result is as follows:

$$\begin{aligned} k_1\left(-\sum_{i\neq 1} b_i + (n-1)\, b_1\right) + k_2\left(-\sum_{i\neq 2} b_i + (n-1)\, b_2\right) + \cdots + \\ + k_n\left(-\sum_{i\neq n} b_n + (n-1)\, b_n\right) = 0 \end{aligned} \tag{3.17}$$

When $k_1 = k_2 = \ldots = k_n$ the equation holds regardless of b_i, showing that the rows of the matrix are not linearly independent and therefore the matrix does not have full rank. This means that it has determinant equal to zero, proving that $\lambda_n = \sum_i a_i - (n-1)$ is an eigenvalue for A.

In order to prove the correctness of the eigenvector $\begin{bmatrix} a_1 - 1 \\ a_2 - 1 \\ \vdots \\ a_n - 1 \end{bmatrix}$, a direct proof is provided in the following:

$$\left[A - \left(\sum_i a_i - (n-1)\right) I\right] \begin{bmatrix} a_1 - 1 \\ a_2 - 1 \\ \vdots \\ a_n - 1 \end{bmatrix} =$$

$$= \begin{bmatrix} -\sum_{i\neq 1} (a_i - 1) & a_1 - 1 & \cdots & a_1 - 1 \\ a_2 - 1 & -\sum_{i\neq 2} (a_2 - 1) & \cdots & a_2 - 1 \\ \vdots & \ddots & \ddots & \vdots \\ a_n - 1 & a_n - 1 & \cdots & -\sum_{i\neq n} (a_i - 1) \end{bmatrix} \begin{bmatrix} a_1 - 1 \\ a_2 - 1 \\ \vdots \\ a_n - 1 \end{bmatrix}$$

And, again by performing the substitution $b_i = a_i - 1$ for the sake of clarity,

it yields

$$\begin{bmatrix} -\sum_{i\neq 1} b_i & b_1 & \cdots & b_1 \\ b_2 & -\sum_{i\neq 2} b_i & \cdots & b_2 \\ \vdots & \ddots & \ddots & \vdots \\ b_n & b_n & \cdots & -\sum_{i\neq n} b_i \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} =$$

$$= \begin{bmatrix} -b_1 \sum_{i\neq 1} b_i + b_1 b_2 + \cdots + b_1 b_n \\ b_1 b_2 - b_2 \sum_{i\neq 2} b_i + \cdots + b_2 b_n \\ \vdots \\ b_1 b_n + b_2 b_n + \cdots - b_n \sum_{i\neq n} b_i \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

The existence of $\lambda_n = \sum_i a_i - (n-1)$ also implies that the algebraic multiplicity of $\lambda = 1$ is actually $n-1$, equal to the corresponding geometric multiplicity.

□

3.2.4 Analysis of convergence time

The results on the eigenvalues and eigenvectors of Subsection 3.2.3 imply that matrix A can be transformed in the Jordan form whose diagonal entries are the eigenvalues of matrix A:

$$A_J = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & \sum_i a_i - (n-1) \end{bmatrix} \tag{3.18}$$

The change-of-basis matrix has the corresponding eigenvectors as columns:

$$T = \begin{bmatrix} 1 & 0 & \cdots & 0 & \frac{a_1-1}{a_n-1} \\ 0 & 1 & \cdots & 0 & \frac{a_2-1}{a_n-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \frac{a_{n-1}-1}{a_n-1} \\ -1 & -1 & \cdots & -1 & 1 \end{bmatrix} \tag{3.19}$$

In order to analyse how the convergence time changes with the eigenvalues – specifically with the eigenvalue λ_n, the only one that changes – the same change-of-basis is applied to the opinion vector $x^{(k)}(t)$, which is transformed to the Jordan-

space opinion vector defined by

$$x_J^{(k)}(t) = T^{-1} x^{(k)}(t) \tag{3.20}$$

The inversion of matrix T is then computed through the procedure reported in [5] that makes use of the Schur complement. In this context, by recurring to the usual substitution $b_i = a_i - 1$ for the sake of clearliness, matrix T is divided into blocks as follows:

$$T = \left[\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & \frac{b_1}{b_n} \\ 0 & 1 & \cdots & 0 & \frac{b_2}{b_n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \frac{b_{n-1}}{b_n} \\ \hline -1 & -1 & \cdots & -1 & 1 \end{array}\right] = \left[\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right] \tag{3.21}$$

where we note that $T_{11} = I_{n-1}$ and $T_{22} = 1$. From [5], we have that

$$T^{-1} = \left[\begin{array}{c|c} (T/T_{22})^{-1} & -T_{11}^{-1} T_{12} (T/T_{11})^{-1} \\ \hline -T_{22}^{-1} T_{21} (T/T_{22})^{-1} & (T/T_{11})^{-1} \end{array}\right] \tag{3.22}$$

with the inverse of the Schur complements $(T/T_{11})^{-1}$ and $(T/T_{22})^{-1}$ computed as

$$\begin{aligned} (T/T_{11})^{-1} &= \left(T_{22} - T_{21} T_{11}^{-1} T_{12}\right)^{-1} = \left(1 + \sum_{i=1}^{n-1} \frac{b_i}{b_n}\right)^{-1} \\ &= \left(\frac{\sum_{i=1}^{n} b_i}{b_n}\right)^{-1} = \frac{b_n}{\sum_{i=1}^{n} b_i} \end{aligned} \tag{3.23}$$

$$
\begin{aligned}
(T/T_{22})^{-1} &= T_{11}^{-1} + T_{11}^{-1}T_{12}(T/T_{11})^{-1}T_{21}T_{11}^{-1} \\
&= I_{n-1} + I_{n-1} \cdot \frac{1}{b_n} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix} \cdot \frac{b_n}{\sum_{i=1}^{n} b_i} \begin{bmatrix} -1 & -1 & \cdots & -1 \end{bmatrix} I_{n-1} \\
&= I_{n-1} - \begin{bmatrix} \frac{b_1}{\sum_{i=1}^{n} b_i} & \frac{b_1}{\sum_{i=1}^{n} b_i} & \cdots & \frac{b_1}{\sum_{i=1}^{n} b_i} \\ \frac{b_2}{\sum_{i=1}^{n} b_i} & \frac{b_2}{\sum_{i=1}^{n} b_i} & \cdots & \frac{b_2}{\sum_{i=1}^{n} b_i} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{b_{n-1}}{\sum_{i=1}^{n} b_i} & \frac{b_{n-1}}{\sum_{i=1}^{n} b_i} & \cdots & \frac{b_{n-1}}{\sum_{i=1}^{n} b_i} \end{bmatrix} \\
&= \frac{1}{\sum_{i=1}^{n} b_i} \cdot \begin{bmatrix} \sum_{i\neq 1} b_i & -b_1 & \cdots & -b_1 \\ -b_2 & \sum_{i\neq 2} b_i & \cdots & -b_2 \\ \vdots & \vdots & \ddots & \vdots \\ -b_{n-1} & -b_{n-1} & \cdots & \sum_{i\neq n-1} b_i \end{bmatrix}
\end{aligned}
\tag{3.24}
$$

The off-diagonal blocks of Eq. 3.22 are computed as

$$
-T_{11}^{-1}T_{12}(T/T_{11})^{-1} = -\begin{bmatrix} \frac{b_1}{\sum_{i=1}^{n} b_i} \\ \frac{b_2}{\sum_{i=1}^{n} b_i} \\ \vdots \\ \frac{b_{n-1}}{\sum_{i=1}^{n} b_i} \end{bmatrix} = \frac{1}{\sum_{i=1}^{n} b_i} \begin{bmatrix} -b_1 \\ -b_2 \\ \vdots \\ -b_{n-1} \end{bmatrix}
\tag{3.25}
$$

$$
\begin{aligned}
&- T_{22}^{-1}T_{21}(T/T_{22})^{-1} = \\
&= \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \cdot \frac{1}{\sum_{i=1}^{n} b_i} \cdot \begin{bmatrix} \sum_{i\neq 1} b_i & -b_1 & \cdots & -b_1 \\ -b_2 & \sum_{i\neq 2} b_i & \cdots & -b_2 \\ \vdots & \vdots & \ddots & \vdots \\ -b_{n-1} & -b_{n-1} & \cdots & \sum_{i\neq n-1} b_i \end{bmatrix} \\
&= \frac{1}{\sum_{i=1}^{n} b_i} \cdot \begin{bmatrix} b_n & \cdots & b_n \end{bmatrix}
\end{aligned}
\tag{3.26}
$$

Joining the blocks of matrix T^{-1}, we obtain:

$$
T^{-1} = \frac{1}{\sum_{i=1}^{n} b_i} \begin{bmatrix} \sum_{i\neq 1} b_i & -b_1 & \cdots & -b_1 & -b_1 \\ -b_2 & \sum_{i\neq 2} b_i & \cdots & -b_2 & -b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -b_{n-1} & -b_{n-1} & \cdots & \sum_{i\neq n-1} b_i & -b_{n-1} \\ b_n & b_n & \cdots & b_n & b_n \end{bmatrix}
\tag{3.27}
$$

Reverting to Eq. 3.20, we obtain the following relation between the opinions in the two basis:

$$x_J^{(k)}(t) = \frac{1}{\sum_{i=1}^n b_i} \begin{bmatrix} x_1^{(k)}(t) \sum_{i\neq 1} b_i - b_1 \sum_{i\neq 1} x_i^{(k)}(t) \\ x_2^{(k)}(t) \sum_{i\neq 2} b_i - b_2 \sum_{i\neq 2} x_i^{(k)}(t) \\ \vdots \\ x_{n-1}^{(k)}(t) \sum_{i\neq n-1} b_i - b_{n-1} \sum_{i\neq n-1} x_i^{(k)}(t) \\ b_n \sum_{i=1}^n x_i^{(k)} \end{bmatrix} \tag{3.28}$$

In the Jordan representation, the first $n-1$ states are associated with eigenvalues $\lambda_1 = \lambda_2 = \cdots = \lambda_{n-1} = 1$, therefore they represent constants of the system, even if the input term $\begin{bmatrix} 1-a_1 & \cdots & 1-a_n \end{bmatrix}^T \bar{u}$ is applied – as it is constant itself. The only state associated with the dynamic of the system is $x_{Jn}^{(k)}(t)$. In order to analyse its characteristics, we transform the input part of Eq. 3.15 through the same change-of-basis:

$$\begin{aligned} B_J = T^{-1}B = T^{-1} \begin{bmatrix} 1-a_1 \\ 1-a_2 \\ \vdots \\ 1-a_n \end{bmatrix} = T^{-1} \begin{bmatrix} -b_1 \\ -b_2 \\ \vdots \\ -b_n \end{bmatrix} = \\ = \frac{1}{\sum_{i=1}^n b_i} \begin{bmatrix} -\sum_{i\neq 1} b_i \cdot b_1 + b_1 \sum_{i\neq 1} b_i \\ -\sum_{i\neq 2} b_i \cdot b_2 + b_2 \sum_{i\neq 2} b_i \\ \vdots \\ -\sum_{i\neq n-1} b_i \cdot b_{n-1} + b_{n-1} \sum_{i\neq n-1} b_i \\ -b_n \sum_{i=1}^n b_i \end{bmatrix} = \frac{1}{\sum_{i=1}^n b_i} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ -b_n \sum_{i=1}^n b_i \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ -b_n \end{bmatrix} \end{aligned} \tag{3.29}$$

The evolution of the n-th state in the Jordan basis is therefore described by the following equation:

$$\begin{aligned} x_{Jn}^{(k+1)}(t) &= \lambda_n x_{Jn}^{(k)}(t) - b_n \bar{u}(t) \\ \frac{b_n}{\sum_{i=1}^n b_i} \sum_{i=1}^n x_i^{(k+1)} &= \lambda_n \frac{b_n}{\sum_{i=1}^n b_i} \sum_{i=1}^n x_i^{(k)} - b_n \bar{u}(t) \\ \sum_{i=1}^n x_i^{(k+1)} &= \lambda_n \sum_{i=1}^n x_i^{(k)} - \sum_{i=1}^n b_i \cdot \bar{u}(t) \end{aligned} \tag{3.30}$$

Introducing the variable $\delta^{(k)}(t) = \sum_{i=1}^n x_i^{(k)}(t) - \bar{u}(t)$, which represents the deviation from the steady state value of the sum of the opinions, we have that $\sum_{i=1}^n x_i^{(k)}(t) =$

$\delta^{(k)}(t) + \bar{u}(t)$ and the update rule becomes

$$\delta^{(k+1)}(t) + \bar{u}(t) = \lambda_n \left[\delta^{(k)}(t) + \bar{u}(t)\right] - \sum_{i=1}^{n} b_i \cdot \bar{u}(t) \tag{3.31}$$

which, by noting that $\lambda_n = \sum_{i=1}^{n} a_i - (n-1) = \sum_{i=1}^{n} (a_i - 1) + 1 = \sum_{i=1}^{n} b_i + 1$, yields

$$\begin{aligned}
\delta^{(k+1)}(t) + \bar{u}(t) &= \left(\sum_{i=1}^{n} b_i + 1\right)\left[\delta^{(k)}(t) + \bar{u}(t)\right] - \sum_{i=1}^{n} b_i \cdot \bar{u}(t) \\
\delta^{(k+1)}(t) + \bar{u}(t) &= \sum_{i=1}^{n} b_i \cdot \delta^{(k)}(t) + \sum_{i=1}^{n} b_i \cdot \bar{u}(t) + \delta^{(k)}(t) + \bar{u}(t) - \sum_{i=1}^{n} b_i \cdot \bar{u}(t) \\
\delta^{(k+1)}(t) &= \sum_{i=1}^{n} b_i \cdot \delta^{(k)}(t) + \delta^{(k)}(t) \\
\delta^{(k+1)}(t) &= \left(\sum_{i=1}^{n} b_i + 1\right) \cdot \delta^{(k)}(t) \\
\delta^{(k+1)}(t) &= \lambda_n \delta^{(k)}(t)
\end{aligned} \tag{3.32}$$

Eq. 3.32 proves that the deviation of the sum of the opinions from their steady-state value defined as $\delta^{(k)}(t)$ evolves as an autonomous system characterised by the eigenvalue $\lambda_n = \sum_{i=1}^{n} a_i - (n-1)$. This means that, at iteration k, its value can be derived from the initial value $\delta^{(0)}(t)$, which depends on the initial opinions of the agents:

$$\delta^{(k)}(t) = \lambda_n^k \delta^{(0)}(t) \tag{3.33}$$

Assuming that we want to consider the number of iterations $\bar{k}$ needed for $\delta^{(k)}(t)$ to drop to the 1% of its initial value and considering the absolute values – as while analysing the convergence time we are not interested in possible oscillations – we have that

$$\begin{aligned}
0.01 &= \left|\frac{\delta^{(\bar{k})}(t)}{\delta^{(0)}(t)}\right| = |\lambda_n|^{(\bar{k})} \\
\bar{k} &= \log_{|\lambda_n|} 0.01
\end{aligned} \tag{3.34}$$

On the other hand, if a maximum number of iterations k_{MAX} for reaching the 1% deviation is desired, we need to introduce an upper bound on λ_n, that is

$$\begin{aligned}
\left|\frac{\delta^{(k)}}{\delta^{(0)}}\right| &= |\lambda_{nMAX}|^{k_{MAX}} \\
|\lambda_{nMAX}| &= \sqrt[k_{MAX}]{0.01}
\end{aligned} \tag{3.35}$$

For instance, considering a maximum number of iterations of 100 for reaching the 1% of the final value, the value of λ_n must not exceed 0.955: following a procedure simular to the one reported in Subsection 3.2.5, this can be enforced by setting upper limits on the weights a_i depending on the number of agents of the system. Considering the definition of a maximum weight a_i^{MAX} equal for all agents, such that $\sum_i a_i \leq n \cdot a_i^{MAX}$, we have that the maximum weight for obtaining $\lambda_n \leq 0.955$ is derived by the following equations:

$$\sum_i a_i - (n-1) \leq 0.955$$

$$n \cdot a_i^{MAX} = n - 1 + 0.955$$

$$a_i^{MAX} = \frac{n - 1 + 0.955}{n} = 1 - \frac{0.045}{n}$$

3.2.5 Enforcing stability by design

Proposition 3.2.2 shows that, in order to ensure that the state-space system is not unstable – which means that the agents asymptotically reach a stable opinion about the quota to be assigned to themselves – the only eigenvalue to be studied is $\lambda_n = \sum_i a_i - (n-1)$, because the eigenvalue $\lambda = 1$ cannot introduce unstable modes as its geometric multiplicity is equal to the algebraic multiplicity.

Thus, the stability condition is defined as

$$-1 < \sum_i a_i - (n-1) < 1$$

Where values -1 and 1 are excluded to avoid permanent oscillations and the possibility of introducing an unstable mode, respectively. Since in the consensus procedure each agent chooses its own value for the self-weight a_i without considering the value chosen by other agents, general bounds must be defined. These are obtained by considering the worst-case scenario where all the agents choose the same self-weight, identifying the lower and upper bounds a_i^{MIN} and a_i^{MAX}. For what concerns the minimum self-weight, it holds that:

$$\sum_i a_i - (n-1) = -1$$

$$n \cdot a_i^{MIN} - (n-1) = -1$$

$$a_i^{MIN} = \frac{n-2}{n}$$

While for the maximum self-weight it holds that:

$$\sum_i a_i - (n-1) = 1$$

$$n \cdot a_i^{MAX} - (n-1) = 1$$

$$a_i^{MAX} = 1$$

Another condition of interest is when the eigenvalue λ_n has a zero value, as this represents:

- The value below which the evolution of the system is oscillatory, which might be an undesired behaviour;
- The value that brings to one-step convergence

If all self-weights a_i are equal to each other, the condition is met when:

$$\sum_i a_i - (n-1) = 0$$

$$n \cdot \bar{a}_i - (n-1) = 0$$

$$\bar{a}_i = \frac{n-1}{n}$$

Therefore, the value $\bar{a}_i$ can be used as the default value for the self-weights, as it is the one that yields the fastest convergence. As for the Exact-Breakdown method, agents are allowed to increase it in case they need to try to make their opinion prevail, at the cost of a slower convergence.

According to the bounds obtained above, the range of possible values for the self-weights is

$$a_i^{MIN} = \frac{n-2}{n} < a_i < 1 = a_i^{MAX}$$

if oscillatory modes are allowed, while if they are not the range is restricted to

$$\bar{a}_i = \frac{n-1}{n} < a_i < 1 = a_i^{MAX}$$

In Table 3.1, values for the bounds a_i^{MIN}, $\bar{a}_i$ and a_i^{MAX} when n changes are reported.

Table 3.1: Allowed values for self-weights when n changes

n	a_i^{MIN}	$\bar{a}_i$	a_i^{MAX}
2	0	0.5	1
3	$\frac{1}{3}$	$\frac{2}{3}$	1
4	0.5	0.75	1
5	0.6	0.8	1
10	0.8	0.9	1
20	0.9	0.95	1
50	0.96	0.98	1
100	0.98	0.99	1

3.2.6 Enforcing hard limits

The constraint imposed by hard limits can be enforced with a method similar to the one used in the Exact-Breakdown method, which is described in Subsection 3.1.4. The proposed strategy involves an additional input $\widetilde{u}^{(k)}(t)$ that agents are allowed to activate when the iteration process is bringing them towards unsustainable quotas. Nevertheless, the way the input is applied to the Self-Focused method is much simpler, as in this case there is no compensation to be applied on the opinion about other agents' candidate quota. In fact, the update rule of Eq. 3.15 is changed to

$$x^{(k+1)}(t) = Ax^{(k)}(t) + B\bar{u}(t) + \widetilde{u}^{(k)}(t)$$

and, with respect to the Exact-Breakdown case, the excesses $u_{i+}^{(k)}$ and defects $u_{i-}^{(k)}$ are changed respectively to:

$$u_{i+}^{(k)}(t) = \left(a_i(t)x_i^{(k)}(t) + (1 - a_i(t)) \sum_{j \neq i} x_j^{(k)}(t) \right) - x_i^{MAX}(t)$$

$$u_{i-}^{(k)}(t) = x_i^{MIN}(t) - \left(a_i(t)x_i^{(k)}(t) + (1 - a_i(t)) \sum_{j \neq i} x_j^{(k)}(t) \right)$$

Also in this case, the Heaviside step function is used to define the input applied by an agent as a function of the excess or defect value, aimed at keeping the

opinion $x_i^{(k+1)}$ at the limit value when the iteration would bring it beyond:

$$u_i^{(k)}(t) = -\theta\left(u_{i+}^{(k)}(t)\right) \cdot u_{i+}^{(k)}(t) + \theta\left(u_{i+}^{(k)}(t)\right) \cdot u_{i+}^{(k)}(t) \tag{3.36}$$

Eventually, the additional input vector $\widetilde{u}^{(k)}(t)$ is composed by stacking all the counteractions performed by the agents:

$$\widetilde{u}^{(k)}(t) = \begin{bmatrix} u_1^{(k)}(t) \\ u_2^{(k)}(t) \\ \vdots \\ u_n^{(k)}(t) \end{bmatrix}$$

As in the Exact-Breakdown case, when the constraint on the total amount defined by Eq. 2.3 is satisfied, it is not possible that all the agents are limited at the same time, therefore convergence is ensured but, also in this case, the activation of the input is an action that slows down the convergence, with consequences that have not been addressed yet.

3.2.7 Usage in MASs represented by sparse graphs

The Exact-Breakdown method exposed in Subsection 3.1 requires that every agent maintains as many opinions as are the number of agents; moreover, agents assign different weights to each of the other agents. This paradigm requires that, during the consensus phase, each node informs every other node about its updated opinions. If we represent the Multi-Agent System with an undirected graph in which nodes correspond to agents and edges correspond to communication channels between the agents, this can be achived in two ways:

- The graph is *complete*, that is, every node is able to exchange information directly with every other node.
- The graph is not complete but connected, and if two nodes are not neighbours (i.e. not connected by an edge), other nodes along a path that connects the two considered nodes take care of the opinion propagation.

Both of the two scenarios introduce some drawbacks: in the first, especially when the number of agents involved is large, the effort required on communication is relevant; furthermore, if some of the communication links suffers from a failure (which means that the graph becomes no longer complete) and if no fallback

mechanism is defined, the mentioned procedure of assigning quotas to agents becomes impossible, because the two involved agents could not update their opinions on each other's quota and share it with all the other agents.

On the other hand, the second option of allowing agents to propagate other agents' opinion requires the introduction of some tight timing protocol, because all the opinions must be propagated to all the other nodes before the following iteration of the update rule occurs. This decreases the robustness of the procedure and can be difficult to apply when the number of not-neighbouring agent pairs grows, thus it was not considered in this study.

In the Self-Focused scenario instead, each agent holds an implicit opinion on the complementary quota to be assigned to all other nodes collectively, and applies the update rule considering the sum of other agents' opinion about their own quota (see the second term of Eq. 3.14). Therefore, agents can perform their update rule by receiving only aggregated opinions consisting of the sum of the other agents opinion. This characteristic of the Self-Focused method was deemed relevant for implementations in which not all the agents are capable of exchanging information with the other agents directly, as the data exchange needed for the propagation of the opinions across the MAS network can be optimized: for example, assume that one agent of a large MAS is neighbour of only another agent, the latter can take care of aggregating all of the other agents opinion through a sum and send only the result to the poorly connected agent. For this reason, the possibility to apply the Self-Focused method to MASs represented by sparse graphs was investigated further. The following paragraphs describe two possible approaches that were considered, which differ by the timing with which the propagation takes place.

Synchronous one-hop propagation

With this approach, during an iteration of the update rule, any data held by an agent can reach only its neighbouring agents. Considering the graph representing the MAS this means that, if two agents n_1 and n_2 have a distance equal to $d > 1$, at iteration k the updated opinion of agent n_2 – that is, $x_2^{(k)}$ – is computed considering the opinion $x_1^{(k-d)}$ of agent n_1 instead of considering $x_1^{(k-1)}$, because it takes d iterations for an opinion of agent n_1 to reach agent n_2. This approach enables to have a fast consensus algorithm, as the time taken to exchange information during an iteration can be considered constant (i.e. the time needed to exchange a message between two neighbouring agents) and does not depend on the diameter

of the graph representing the MAS. Of course, the introduced delay changes substantially the update rule of the agents and thus the dynamics of the system: most importantly, the criteria for stability reported in Section 3.2.3 must be verified. The review of the stability criteria was considered beyond the scope of this project, but the implementation of such a technique was nevertheless tested using the same stability criteria exposed in Section 3.2.3. The details of this implementation are reported in Subsection 4.2.3 and Appendix A: although the theory behind the dynamics of such kind of MAS is yet to be formalized, the empirical results obtained are promising as multiple tests did not highlight an unstable behaviour, although the quotas assigned with this procedure are slightly different from those obtained in case the graph representing the MAS is complete, for a reason that is detailed in Subsection 4.2.3.

Multi-hop propagation

With this approach, the whole opinion propagation process happens in the time window between two subsequent iterations of the consensus algorithm. With respect to the one-hop propagation approach, this means that the consensus algorithm is slowed down because between two iterations there must be an amount of time that is sufficient for an opinion to traverse a number of hops equal to the graph diameter, making this drawback more relevant when considering larger and sparser MASs. In practical terms, this also means that agents must take care to propagate opinions at a faster rate with respect to the rate at which they compute their own opinion updates, requiring the two activities to be handled independently, while with the previous approach a single program cycle could fulfill them both. On the other hand, from the consensus algorithm standpoint, all the theoretical results exposed in this Section apply directly: this kind of propagation implies that, at each iteration of the consensus algorithm, every agent is aware of the sum of all the other agents' opinion at the previous iteration as if the graph representing the MAS was complete. In fact, with this approach the opinion propagation process can be considered something that happens in background, not impacting at all the update rule execution. Figure 3.1 shows the difference between the one-hop and the multi-hop propagation techniques. The multi-hop propagation was not tested because it was considered less interesting than the Synchronous one-hop propagation, as the former would be completely equivalent to the complete graph scenario for what concerns the dynamics of the opinions.

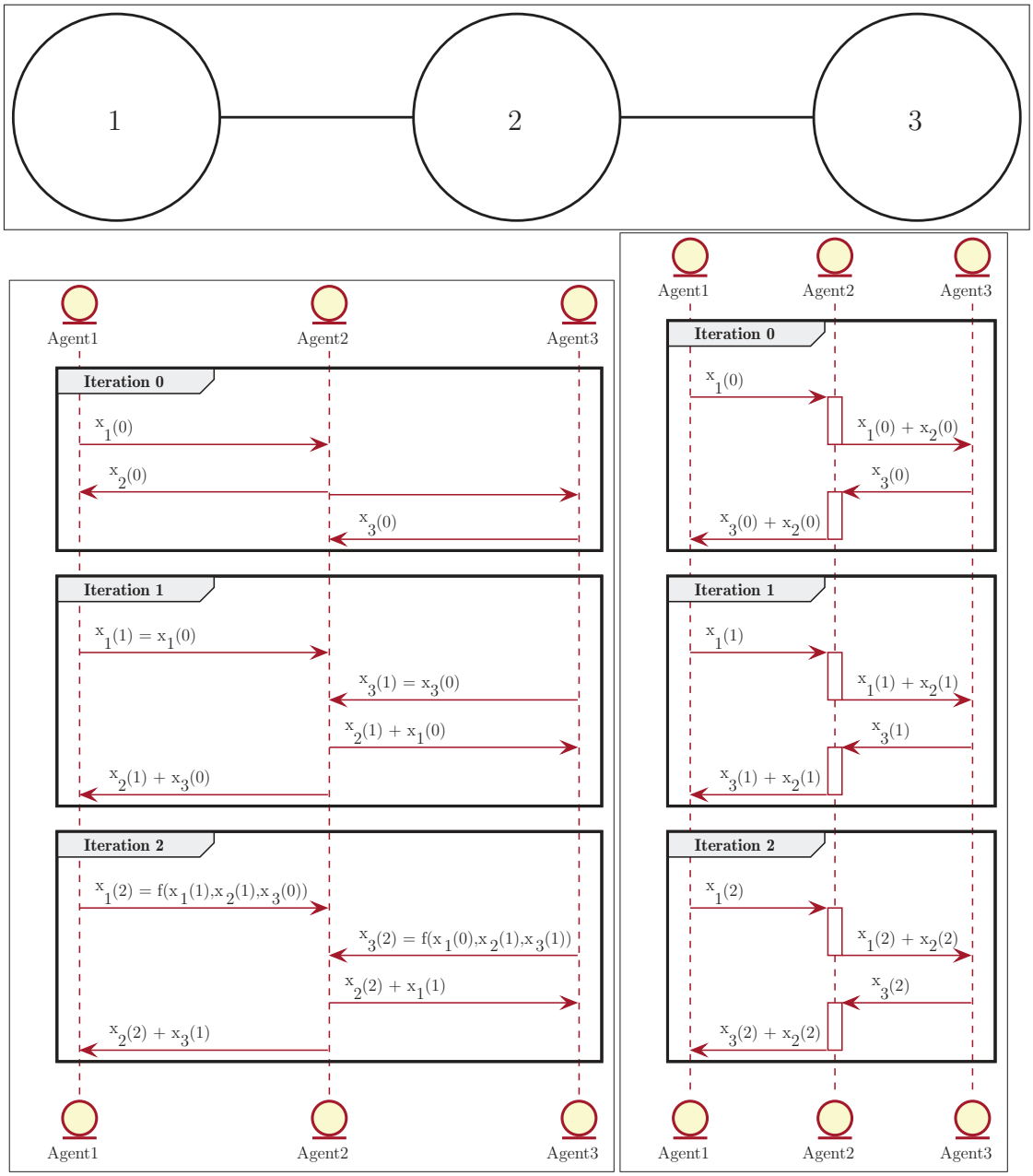

Figure 3.1: Example of differences between the *Synchronous one-hop propagation* and the *Multi-hop propagation*. At the top, the graph representing the MAS; at the bottom, Unified Modeling Language (UML) sequence diagrams showing the one-hop propagation (left) and the multi-hop propagation (right).

Chapter 4

Implementation

After the definition of the theoretical method to address the allocation of quotas was completed, obtaining the results reported in Chapter 3, a way to implement the method was sought. The implementation described in this Section is related to the Self-Focused method only, as it was considered the one with the wider range of applications, but most of the techniques involved could be reused for the implementation of the Exact-Breakdown method, requiring only changes to the part of the code that actually performs the mathematical computation of the iterations. More generally, the author thinks that many aspects reported in this Section are relevant for the implementation of a wider range of Multi-Agent Systems: for this reason, the code base developed for this thesis is available in a public Git repository located at `https://gitlab.dei.unipd.it/alviserossi/thesiscode`, in the hope that it is useful for other similar projects.

The design of the implementation was driven by the following two main requirements:

- **The developed code base shall be meant for both simulation and for real-world applications**: this means that the code can be written and tested without working on some specific hardware – especially for what concerns the tests on the interaction among agents, which shall not require the availability of many devices. Nevertheless, porting the code used for simulation to a specific hardware environment would require some unavoidable additional development, mostly related to the management of I/O for the specific platform. This requirement is hard to match when using a typical academic environment like Matlab for simulation, as in that context it is not easy to run in a parallel fashion the same agent code multiple times and make every of these instances interact with each other. Moreover, the code developed

in Matlab would hardly be portable to an actual agent environment, requiring the code to be rewritten and tested before being able to deploy it in a real-world application.

- **The developed code base shall be meant for a wide range of hardware platforms**: as the problem of allocation of quotas has many real-world applications, the code base should be capable to run on the widest range of hardware devices, ranging from industrial servers to IoT devices.

The two requirements mentioned above led the author to choose a development environment based on Docker Containers, for reasons that are explained in the following subsection.

4.1 Usage of Docker Containers

Containerization is a technique that gained more and more consideration during the last decade as it was an important ingredient for the migration of many IT services to cloud environments, though for this implementation it was chosen for different reasons – mainly because it allowed to simulate an IP network of agents on a single computer and because it allows the code to run on many kinds of hardware and operating systems, as will be detailed later in this section. A brief overview of what containers are and the advantages they introduce can be found in at [6], where they are referred to as "a new way to abstract one or more processes from the rest of a system. [...] They also provide a standard way to package and isolate application code, configurations, and dependencies into a single object". Containers are different from a virtual machine as they do not include an operating system, they rely on the operating system of the machine where they run – the *host* operating system, which makes them much more lightweight than virtual machines. By default, containers are isolated from the host operating system, which means that they cannot access any resource available on its disk drives or connect to its network: specific exceptions must be configured if a container requires them.

Although the concept of containers was known before, their usage was boosted in 2013 with the launch of the Docker Engine, a software used for creating, managing and running containers. Docker allows to create and use *container images*, which are software packages that are used to execute one or more instances of a container (possibly specifying configuration parameters that define the behaviour

of the container, if the bundled software allows to). Container images can be used also to build another image that expands the functionality of the original one: for instance, a developer can retrieve the publicly available image of an interpreter such as Python, include the Python code developed for a specific application and build a new image that includes both the code and the Python interpreter, which allows the code to be executed directly on an operating system that does not have the Python interpreter installed. A similar example is the one in which the developer needs to build a container image that contains a simple website: for such a need, one can start from the image of a generic web server – there are many publicly available ones such as Apache – and include web pages developed for the specific website, then build a new self-consistent image with the web server application and the custom web pages. Publicly available images can be found in online repositories such as the *Docker Hub*.

Docker allows to execute the following operations that were used in the implementation:

- **Build**: create an image from the source code, possibly including another container image

- **Run**: execute a container instance, possibly passing parameters to the inner software and specifying the exceptions to the isolation from the resources of the host that runs the Docker engine, which could be needed for the proper behaviour of the software. Two examples of exception to the isolation are reported below:

  - TCP or UDP ports of the container that must be accessible from outside the container environment, which are used for exchanging data with other applications or users – for instance, referring to the example of the website reported above, a TCP port must be exposed by the container in order to transport the HTTP protocol traffic and serve requests from the browsers connecting to the website

  - Storage volumes: for containers that need data persistence, such as a container that implements a database server, a directory of the host system can be made accessible (i.e., *mounted*) to the container, so that if it is restarted there is no data loss.

- **Compose**: the *Docker Compose* feature allows to launch a group of containers at once. It can be used to automate the execution of an environment

composed by many containers through the definition of a text file (namely the *Compose file*) that specifies which containers must be launched and their configuration. In this study it was used for simulation purposes, as the simulation consisted in launching a number of containers representing the agents of the Multi-Agent System and some auxiliary containers useful for the simulation, as will be shown later.

One useful aspect of the Compose feature is that it allows *scaling*: starting from the same Compose file that defines the group of containers to be launched, when the environment is launched users can specify how many instances they want to launch for each of the single containers reported in the Compose file. Although the feature is arguably meant to be used for load balancing among many containers in IT applications, in this project it was used to be able to quickly change the number of agents that the Multi-Agent System is made of: in the Compose file only one agent container was reported, then the number of agents for each simulation was defined leveraging on the scale feature.

As previously stated, one of the two main reasons why the author chose to use Docker containers for the implementation is that, when many containers are deployed onto the same host, they are allowed to communicate with each other over a virtual IP subnet. A default configuration for the network can be used, otherwise the subnet parameters such as the address range can be customized through commands or configuration files. By default, this virtual network is isolated from the network of the host (that is, containers can communicate only with each other), but specific container TCP or UDP ports can be made accessible from the host network, as was previously reported.

Each of the containers is assigned an IP address in the virtual subnet range, that can be dynamically assigned or manually configured for each of the containers. A specific feature of the IP protocol that is available in the virtual subnet (as in actual IP networks) is the capability of sending *broadcast packets*, which are packets that are received by all the members of the network – at least in the same subnet, while when routing is involved broadcast packets are forwarded to other subnets only if the routers along the path are configured accordingly. Broadcast packets were used to implement a simple *discovery* protocol that allowed the agents of the network to be aware of each other's existence with a very low effort, regardless of the number of agents involved and the IP address that they were assigned in a particular scenario.

The availability of the virtual subnet made it possible to simulate the protocols defined for the communication among the agents of the MAS exactly as if the code was running on different hosts of a physical LAN, which reduces the efforts required to port the code in a real-world scenario. In the following section, a description of the container types that were developed and the definition of their communication protocols is detailed.

4.2 Container specifications

In this section the different container images that were implemented are described. For the simulation purposes, four image types were considered, one related to the software running on an agent and three related to auxiliary services. All these images could be used also in real-world applications with little changes to the code.

- **Agent**: the image containing the software that is meant to run on an agent of the Multi-Agent System. The code implements the iteration logic needed for reaching consensus among agents; moreover, the code handles the communications with other agents (that is, other instances of the same container image) and with the other container types reported in the following. For simulation purposes, this image type is instanced a number of times equal to the amount of agents that the simulation requires; for real-world applications, a single instance of the image must be deployed onto the hardware associated with each of the agents involved in the MAS. For simulation purposes, more than one image of this kind was developed in order to test different behaviours of the agents (see Chapter 4), but they all represent the code that runs on an agent of the network.

- **Gateway**: the image containing the software that acts as a single point of contact between the agent network and the host network for real-time operations. It takes care to receive from the user the commands to be issued to agents and forward them to the involved agents; it also collects real-time information about the state of the MAS and makes it available to the user. This image and the following two are meant to be deployed only once in the virtual subnet.

- **Coordination dashboard**: an image that exposes a simple web page that the user can access through the browser. The web page exposes a user

interface and makes the connecting browser download the code that allows it to interact directly with the gateway. Therefore, it plays no role in the actual communication with the agent network: it only provides the user with the means needed to interact with the gateway. In order to simplify the architecture, the gateway and the coordination dashboard container images could have been aggregated in a single container that implements both features, but this was not done because it was not a quick task and it did not introduce relevant benefits in terms of features.

- **Influx**: an image containing an instance of InfluxDB, an open source time-series database that is available online. This container is used to store the data produced by the agents so that it can be retrieved and analysed as the result of the simulation. Nevertheless, an instance of InfluxDB could be deployed also in a real scenario for the same purposes.

In order to start a simulation, the four types of container are launched by means of the Docker Compose feature described in Section 4.1, specifying the number of agent instances to be launched through the `scale` parameter of the `docker compose up` command. Figure 4.1 shows a diagram of the resulting networks and their connected members.

The rest of this section reports an overview of the designed communication protocols and of the programming languages that were used for the implementation. The details about the structure of the code included in the container images is reported in Appendix A.

4.2.1 Communication protocols

Any kind of information exchange involved in the project is based on TCP connections and UDP datagrams, involving both the IP subnet that the agents are connected to (either virtual or real) and the possible outer network that the user is connected to. For the services built on existing applications such as the Apache web server or the one used by InfluxDB, the default port numbers were used. For the services that were defined ad-hoc for this project, a port number belonging to the non-dynamic range and not already registered for well-known services had to be defined: both for TCP and UDP based messages, the choice fell on number 21852[1].

[1]The port number was chosen as a tribute to Joe Strummer (born August 21st, 1952), frontman of *The Clash*.

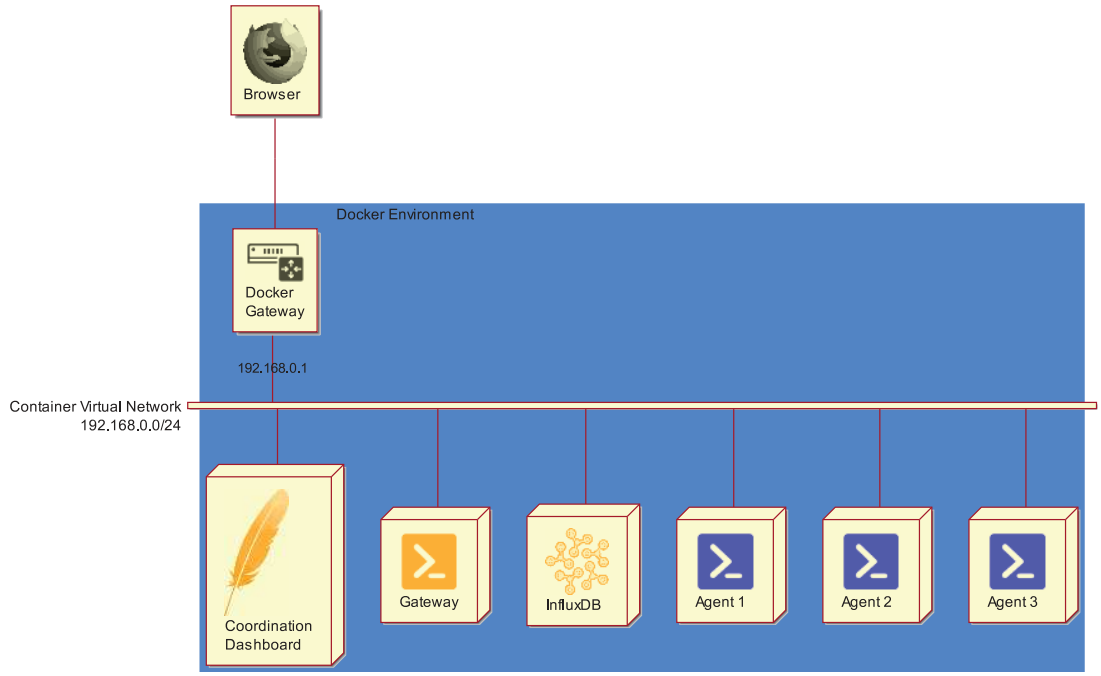

Figure 4.1: Network diagram of the simulation environment deployed through the *Docker Compose* command, with a number of agents equal to three. Components represented as a cube are the deployed contaners: the ones with the orange logo are the service containers, the ones with the blue logo are the agent containers.

The messages that the containers need to exchange are grouped into three categories, detailed in the three following paragraphs.

Discovery messages

The discovery messages are the ones through which each agent informs the whole MAS of its existence. As anticipated before, for this kind of messages it was chosen to leverage on the capability to send a broadcast packet over an IP network, which allows to send a packet that is received by all the members of the network. For this purpose it is not possible to use the TCP transport protocol, as its property of being connection-oriented does not allow for one-to-many communications. Therefore, discovery messages were implemented through a simple UDP datagram whose payload reports the hostname of the sending agent: upon reception of such a datagram, a member of the network is able to update a list of *known agents*, reporting the map between agents' hostname – used as unique identifiers of the agents – and the IP address currently assigned to each of them, which is determined from the UDP packet metadata, specifically from the sender IP address. The IP address was not directly used as an identifier for the agents as

there might be situations in which an agent changes its IP address, for instance after a temporary network disconnection, and using the IP address as an identifier would not allow to correlate the same agent's data before and after such an event. Moreover, the *known agents* list is the basis upon which each agent knows the number of agents available in the network, an information needed to define the acceptable range of weights that ensures the stability of the consensus procedure, as reported in Subsection 3.2.3.

The broadcast message is not only received by the agents: it is also received by the gateway container, which maintains its *known agents* list as well, and uses it to both report the list of available agents to the user browser upon need and to know which IP addresses it needs to use in order to forward the commands coming from the user browser.

The usage of UDP as the transport protocol has a main drawback: UDP does not implement any reliability feature as retransmissions and error check, which are natively implemented when using TCP. Anyway, discovery messages are periodically resent and, if an agent never succeeds in informing another agent of its existence, the reason is probably related to a serious network issue that would nevertheless make the communication between the two agents impossible, therefore it is acceptable that the two are not informed of each other's existance. Moreover, possible discovery techniques based on TCP such as attempting to open TCP connections towards all the addresses of the subnet were considered inefficient and not scalable to large networks, making the usage of UDP the only choice.

Core messages

After the IP address of each agent is known to the network members thanks to discovery messages, members have all the information needed to exchange one-to-one messages. In this scenario, the increased reliability of TCP was considered a relevant advantage as it requires much less effort for handling network issues when developing the application code. Moreover, HTTP was chosen as the application layer protocol and the HTTP payload was formatted in JSON: these two choices combined represent common practices in the implementation of RESTful APIs, which are a kind of web service used to exchange data structures over the Internet that is nowadays widely used. Their wide usage was the driver of the design choice, as it means that many runtime environments have built-in features to handle this kind of data exchange very easily. Using such a method is arguably

not the best choice in terms of efficiency, as HTTP does not natively implement compression of the payload and introduces overheads that might be avoided using other protocols, but for the sake of this project these downsides were considered acceptable.

A positive effect of using JSON for the payload format is that it allows to easily define *schemas*, a property shared with other data structure formats such as XML. A JSON schema represents a definition that specifies how a JSON message must be built to be considered acceptable for that kind of message, including constraints about mandatory data and the data type of the values reported in the message. A JSON schema is itself structured in JSON format, and modern programming languages have built-in tools to easily test a JSON message against a JSON schema, allowing the detection of malformed messages without the need of developing custom code for the purpose.

The group of core messages includes the ones that were designed specifically for the purpose of the project. In the following, the list of messages belonging to this group is reported:

- **Total amount**: the message used for notifying agents of the total amount to be broken down into quotas. The only mandatory variable of this message is the total amount value; optionally, also the time range in which that total amount is valid can be reported. This allows a future implementation in which consensus can be reached not only for real-time allocation of quotas, but also for quotas valid for the future, resulting in the capability of agents of reaching a consensus on the quotas to be taken care in advance. In the simulation environment, the message is firstly sent by the user browser to the gateway container when the user clicks on the associated *Send* button of the User Interface; the gateway then takes care of forwarding the command to all the agents available on the network.

- **Iteration**: the message that an agent sends to every other agent as soon as it computes a new opinion about the quota to be assigned to itself. It reports the opinion value, the iteration index (so that the receiving agent is able to ensure it did not miss any message after the previously received one, an event that would have impacts on the consensus procedure) and the hostname identifying the agent that computed the opinion. The last information was included to pave the way for an implementation on agent networks where multi-hop message transmission is required as not all the agents are capable to exchange information with each other: in that context,

this additional field allows an agent to transmit an opinion on behalf of another agent.

The iteration message is only exchanged between the agents of the MAS.

Auxiliary messages

These are the messages that are not strictly related to the implementation of the consensus algorithm, but are used for operation of the MAS and analysis of the data produced by the agents. They include:

- **Get agent list**: The request that the coordination dashboard sends to the gateway to retrieve the list of agents currently available on the network. Implemented through the same set of protocols of the core messages – JSON payload on an HTTP request served on TCP port 21852.
- **Save data**: The message through which agents store data onto InfluxDB. The standard API defined by InfluxDB is used for the purpose.
- **Exit**: The message involved in the procedure used to gracefully stop the software running on the agents. The message is primarily sent from the coordination dashboard to the gateway as an HTTP request without payload on a specific URL; upon reception of such a request, the gateway forwards it to each of the agents. As will be shown in the following, since the implementation is based on parallel processing, the message is then further forwarded by the main process of the agent software to its auxiliary processes: in order to stop the auxiliary process that handles UDP messages, the message transport protocol is switched to UDP.

Figure 4.2 and 4.3 offer a graphical representation of all the messages involved in the operation of the MAS.

4.2.2 Programming languages and development techniques

The implementation activity involved two different kinds of software development: the less challenging one was related to the Coordination dashboard, the webpage allowing the user browser to interact with the gateway, which was implemented through HTML and Javascript. HTML was used to define the static part of the webpage, including textboxes where the user can type numeric values for commands and buttons to launch actions. The Javascript part is the one that makes

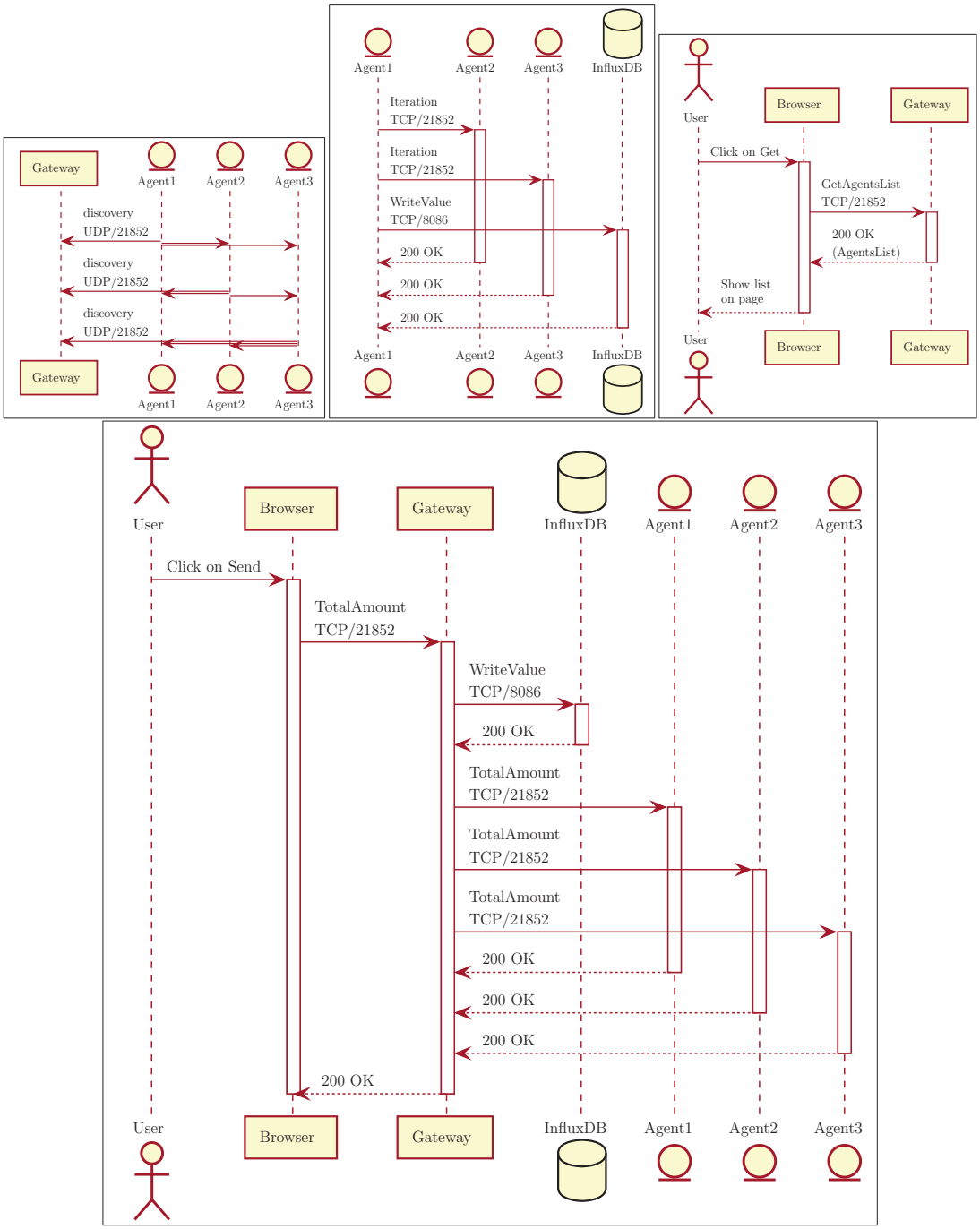

Figure 4.2: Unified Modeling Language (UML) sequence diagrams representing message exchanges among the member of the networks for a network of three agents. The top-left diagram represents the messages exchanged for agent discovery; the top-middle diagram the messages that *Agent 1* sends to the other members in order to inform about its updated opinion during the iterations of the consensus process; the top-right diagram represents the sequence of messages that allow the user to retrieve the list of agents in the network; the bottom part represents the message sequence associated with a Total Amount command sent by the user.

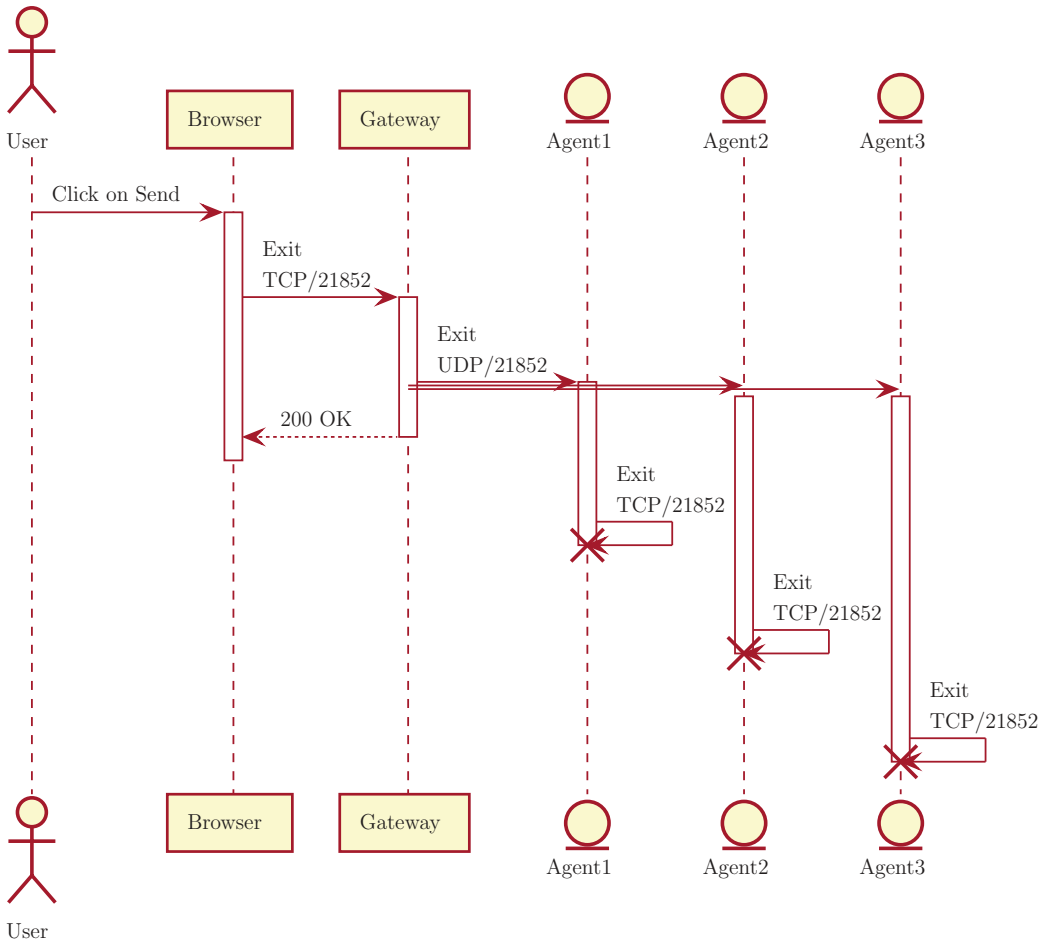

Figure 4.3: Unified Modeling Language (UML) sequence diagram representing the message exchanged when the user wants to stop the operation of a network of three agents. The self-pointing arrows on the agents part represent network packets needed to gracefully terminate the TCP listener process of each agent.

the user browser interact with the gateway: when the user clicks on a button, a Javascript function that sends the command to the gateway is launched. For commands that return an output to the dashboard, such as the *Get agent list* one, the same Javascript function also takes care of changing the displayed webpage in order to show the command output to the user. As the Coordination dashboard consists of a single webpage which is dynamically changed by code running on the client side, the technique used is loosely related to *AJAX* (acronym for Asynchronous Javascript and XML), a web development paradigm that consists in designing background data exchanges between the browser and the server, with the Javascript code running on the browser taking care of handling the data exchange for the user and updating the page aspect according to the outcomes of the data exchange. This paradigm emerged during the last two decades as a more efficient alternative to the conventional interaction between a browser and a

webserver, in which the browser requests a webpage to the server, the server optionally runs some code to customize the page for the specific request and returns a static webpage to the browser, then any new operation that the user might require must pass through a whole new page loading process. More details about the Coordination dashboard development are reported in Appendix A.

The other kind of software development involves the code running on the agents and the one of the gateway. For these purposes, any kind of general purpose programming language that allowed containerization could be used. The choice fell on PowerShell, an interpreted scripting language developed by Microsoft that is primarily meant to be used for automation of IT system configurations, which can nevertheless be used to develop general purpose simple applications. The choice was mostly related to the fact that PowerShell was the language the author is most experienced with; moreover, it has the advantages of being cross-platform (although it was originally built for Microsoft Windows systems, since 2016 version 6 was released with runtime environments for Linux and macOS operative systems) and already present in all Windows installations, although the available PowerShell version changes with the version of Windows, requiring some attention on compatibility aspects. An official Docker image of PowerShell is publicly available. The main drawback of using an interpreted language is on the performance side, as compiled languages perform better due to the lower effort required to the hosting system at runtime, but for the implementation of the consensus algorithm the computational burden is not relevant, making the usage of an interpreted language acceptable. Moreover, purely compiled languages are usually less interoperable than interpreted ones, as compilation is an operation that is specific to the operative system and hardware where the code needs to run.

Although the PowerShell language was already fairly known to the author, some advanced techniques were learnt during this project. A useful source that was used for acquiring the related knowledge was [7], whose chapters range from some that allow to learn the basic concepts of the language to those describing the most advanced features available. The source was especially used for understanding how to handle parallel processing: in particular, the software running on the agents had to be capable of listening to messages that might be received at any time from other members of the network and react to these messages. The most basic instructions that allow a PowerShell script to listen on a network are meant to be used for *synchronous* operations, which means that the code execution halts

after the instruction related to the message reception, until a message is actually received. Using such an approach would have meant that the program would have not been capable of handling the reception of the UDP discovery messages and the TCP core messages concurrently, nor that it could perform the computations related to the consensus algorithm iteration while listening to any kind of message. Instead, in order to implement an *asynchronous* agent message reception, the author opted for the usage of *Background Jobs* described in Chapter 13 of [7]. The concept behind this implementation is that the main program running on the agents performs the actions described in Algorithm 1, starting background jobs and periodically checking if they produced some output. Another task where

Algorithm 1 Agent code structure

```
START discovery_message_sender_job
START udp_listener_job
START tcp_listener_job
while exit_signal = false do
    READ udp_listener_job output
    if UDP messages received then
        UPDATE known_agents_list
    end if
    READ tcp_listener_job output
    if TCP messages available then
        STORE total_amount, other agent opinions or exit_signal
    end if
    if total_amount available and consensus algorithm not initialized then
        k ← 0
        COMPUTE initial_opinion
        COMPUTE weight
        SEND initial_opinion
    end if
    if all opinions received for iteration k then
        COMPUTE updated_opinion
        SEND updated_opinion
        k ← k + 1
    end if
    WAIT pause_time
end while
STOP all jobs
```

parallel processing was used is the one related to spreading the agent opinion: the related message is sent through HTTP requests directed to each of the other agents, each of which can take some tenths or hundreds of milliseconds to be

completed (although the time is mostly spent because of network latency and while waiting for the other agent's response, without relevant computational burden). Therefore, using the standard synchronous instruction for issuing HTTP requests would have meant to serialize these requests, resulting in a time spent for spreading an agent opinion proportional to the number of agents in the network, a drawback particularly relevant in large MASs. Instead, HTTP requests were issued in parallel to all the agents using PowerShell *Runspaces* (widely described in Chapter 20 of [7]), which are implemented through separate threads of the main PowerShell process, as in this case using Background Jobs would introduce a heavier overhead because they run on separate processes – furthermore, in this case the capability of Background Jobs of being able to produce an output for the main process was not needed, as the only thing the additional thread has to take care of is to send a single HTTP request.

4.2.3 Implementation on sparse MASs

In order to apply the consensus algorithm on Multi-Agent Systems represented by sparse graphs, according to the considerations expressed in Section 3.2.7, three main challenges must be faced:

- How to modify the network protocols described in Section 4.2.1 to simulate a sparse MASs scenario;
- How to modify the discovery process so that every agent is made aware of every other agents' existence and of the path that must be used to exchange messages;
- How to handle the opinion propagation across the MAS network, especially for what concerns the situations in which a set of opinions can be aggregated into their sum and the aggregated opinion can be propagated.

These three topics are addressed in the following paragraphs.

Simulation of a sparse MAS

The communication protocols described in Section 4.2.1 are consistent with a scenario in which every member of the network can exchange direct communication with each of the other network members, that is, one in which the graph representing the network is complete. More specifically, the discovery messages that every agent sends to inform the other agents of its existence are sent in

broadcast on the whole IP subnet, which means that they are received by all the network members, and when an agent needs to inform another agent of its updated opinion, the former opens a direct TCP connection towards the latter. Instead, in order to represent a sparse network of agents in simulation, a different method must be followed. The chosen method was driven by the following design requirements:

- Each simulation run should be based on a different random sparse graph, so that the validity of the consensus algorithm in the multi-hop case is verified on the larger number of topologies possible;
- The implemented method should allow to influence the sparsity of the randomly-generated graph through some tuning parameter, in order to be able to observe how the algorithm behaviour changes with the sparsity of the graph;
- The topology of the graph should not be defined by a centralized system external to the agent containers, as this would decrease the distributed property of the environment and increase the differences between the simulation and a possible real-world scenario. The topology of the graph should be defined somehow by the agents themselves;
- As for the other design choices described in this Chapter, the implemented method should require the least changes possible when it needs to be applied to a real-world scenario.

The design requirements reported above led to the usage of *multicast groups*, where the term *multicast* represents the fact that they involve a subset of network members, as opposed to *broadcast* communications in which all the members of a network are involved. Multicast groups are a feature of the IP protocol that allows network members to *join* a multicast group defined by a *multicast address*: when a network member joins a multicast group, from that moment onwards all the messages sent to the associated multicast address are delivered to it, and to all the other network members that joined the same multicast group. All the IPv4 addresses ranging from 224.0.0.0 to 239.255.255.255 are dedicated to multicast groups, with subranges assigned for particular purposes[2]. For the implementation described in the following, the range from 224.0.224.0 to 224.0.229.255 was

[2]For more details, see `https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml`, the page where the *Internet Assigned Numbers Authority* publishes information about multicast addresses and their subgroup purpose.

chosen, which is a portion of the multicast address space that is not assigned to any specific purpose yet: for example, this means that when the simulation requires the availability of $n \leq 255$ multicast groups (they are considered enough for simulations involving a reasonably large MAS), the ones with address from 224.0.224.1 to 224.0.224.n are used.

As reported in Section 4.2.1 with reference to broadcast messages, the TCP protocol is intrinsically connection-oriented, therefore it does not allow one-to-many communications such as the ones provided by multicast addresses: for this reason, the multicast-based communications defined in this study always rely on UDP as the transport protocol.

In the context of this study, the usage of multicast groups boils down to the definition of whether two existing agents of the simulated MAS are neighbours. A customizable number of multicast groups m is set before the simulation begins and, when the code of an agent starts to run, it immediately chooses whether to join each of the single multicast groups available (possibly joining more than one group, or no group at all): then, all the other agents of the network that share at least one multicast group with the considered agent represent neighbouring agents. By analogy, multicast groups can be seen as radio channels that are available for the agents to exchange information, with each agent listening and sending data only on a specific subset of radio channels: if two agents share the connection to at least one common radio channel, they are allowed to exchange information directly; otherwise, they could still be able to communicate indirectly, by leveraging on a third agent that shares the connection to a radio channel with each of the two former agents separately. In the latter case, the third agent can take care of forwarding the message to be exchanged between the two former agents. Of course, the same concept applies to paths that involve more than one forwarding agent. Figure 4.4 shows an example of graph corresponding to an association between each agent and the multicast group it joined.

As one requirement of the simulation is that the generated sparse graph has a random topology, the agent code developed for the simulation chooses at random which are the multicast groups to join. More specifically, the code considers the $m \geq 1$ available multicast groups separately and, for each of them, extracts an outcome from a Bernoulli random variable with parameter $0 \leq p \leq 1$, p being constant for all the agents; if the outcome is 1, the agent joins that multicast group, if it is 0 it does not. Referring to the requirement of allowing to influence the sparsity of graphs through some tuning parameters, with this approach

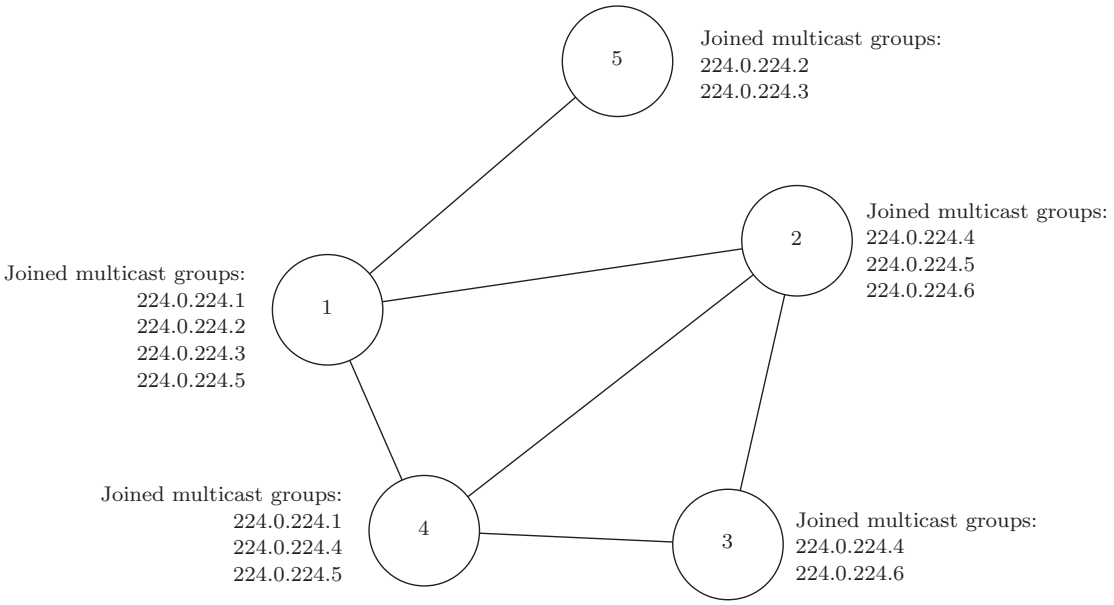

Figure 4.4: Example of sparse graph representing the association between each agent and the multicast group it joined.

both the defined amount of multicast groups m and the Bernoulli coefficient p have an influence: as the involved extractions are independent and identically distributed, considering a single multicast group, the probability associated with the event that two agents can communicate through that multicast group is p^2, therefore the probability associated with the event that they cannot communicate through that specific multicast group is $1 - p^2$; the probability that two agents cannot communicate through any of the available multicast groups is therefore $(1 - p^2)^m$, which leads to the final definition of the probability that two agents are neighbours, equal to $1 - (1 - p^2)^m$. As expected, the formula shows that an increase in either the Bernoulli coefficient p or the number of multicast group m brings to a higher probability of two agents being neighbours. Of course, a higher probability for each pair of agents of the graph to be neighbours means a lower extent of sparsity for the resulting graph. Figure 4.5 represents the probability that two agents are neighbours as a function of p and m. Another approach for simulating a random topology could have been to make every agent choose whether to connect or not to any other specific agent, without relying on multicast groups but again using the outcome of a Bernoulli random variable. This

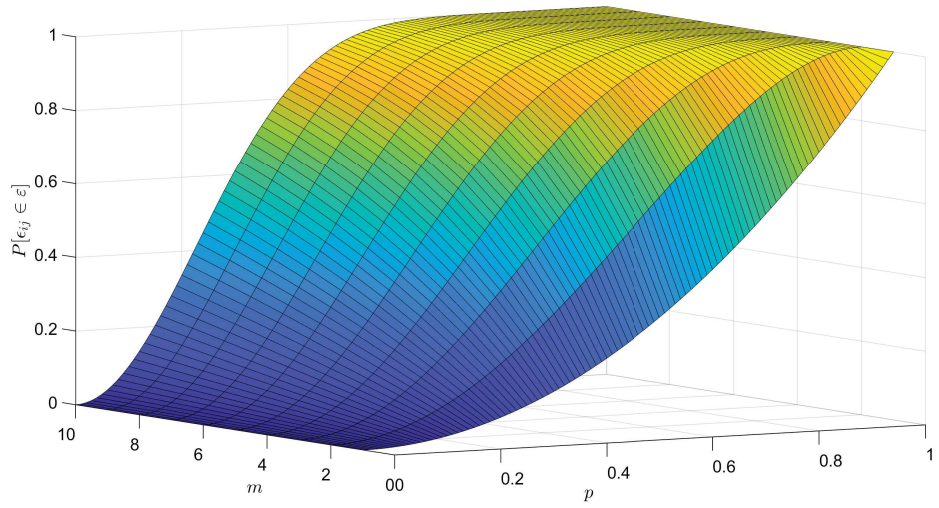

Figure 4.5: The probability of the events that two agents are neighbours as a function of the Bernoulli parameter p and the number of multicast groups m, for $1 \leq m \leq 10$.

would have required a preliminary step in which a full discovery is performed, during which every agent becomes aware of the existence of every other agent of the MAS, then the definition of the edges of the graph would have taken place through the mentioned process. Besides the addition of the preliminary discovery phase, the solution was discarded because it requires a number of extractions that grows with the square of the agents of the MAS, as opposed to a number of extractions proportional to the product between the amount of agents and the amount of multicast groups (which in large MASs is meant to be lower than the number of agents); moreover, the usage of multicast groups is a technique that could be adopted also in real-world scenarios, where the list of multicast groups that an agent needs to join could be defined through a meaningful logic instead of a random one, allowing to quickly define the desired topology of the MAS.

It is worth to stress that, by using multicast groups, the partition of the agents is only a logical grouping; technically, all the agents are still connected to the same subnet and could exchange messages directly. In fact, after the discovery phase that defines the graph topology and leverages on multicast groups, neighbouring agents exchange messages by means of direct TCP connections; from the network perspective, they could do the same also with non-neighbouring agents, but the discovery process described in the following paragraph provides them with information useful to interact directly with the neighbouring agents only.

Discovery process for sparse graphs

While the discovery process reported in Section 4.2.1 consists of a simple message containing the hostname of the sender that is spread to all the other agents by means of a broadcast packet, additional information needs to be exchanged between the agents when the graph representing the MAS is not complete. This is due to the fact that each agent needs to become aware not only of the existence of its neighbours, but also of the existence of all the other agents, in order to make sure that the choice of the weight it performs is compatible with the amount of agents composing the MAS, as defined in Section 3.2.5. Moreover, as there could be more than one path connecting two agents of the MAS (and there usually is), a way allowing an agent to become aware of the overall amount of agents in the MAS without maintaining a list of identifiers for each of them was not found. Therefore, the goal of the discovery process for the sparse MAS case was defined as making each agent maintain an **agent map**, which consists of a list of tuples, each one related to a remote agent of the MAS. Using the term *local agent* to define the agent holding the specific agent map and *remote agent* to define the agent described by one specific tuple, each of the tuples contains the following fields:

- The hostname of the remote agent
- The hostname of the next-hop agent (neighbour of the local agent) along the path between the local and the remote agent. If the remote agent is a neighbour of the local agent, this field contains the hostname of the remote agent as well
- The IP address of the next-hop agent, needed to establish the network connections. If the remote agent is a neighbour of the local agent, this fields contains the IP address of the remote agent
- The distance between the local agent and the remote agent in terms of hops. If the remote agent is a neighbour of the local agent, this field is set to one. This field is used in case more than one path is available between the local and the remote agent, to select the next-hop agent that ensures the shortest path to the remote agent.

Because of the way the discovery process work, the agent map includes also a special tuple representing the local agent, with next-hop fields set to the placeholder *null* value and the distance set to zero. The discovery process that makes

every agent obtain the mentioned fields for all the agents of the MASs consists of the following steps:

1. At the beginning of the process, every agent stores in its agent map only the special tuple about itself.

2. Every agent periodically shares its agent map with all of its neighbours, that is, it sends the information to the multicast addresses associated with the multicast groups it joined.

3. Upon reception of an agent map from a neighbour, an agent increases all the distance values reported in the received tuples by one in order to account for the additional step between the neighbour and the local agent. Then, the local agent checks whether each remote agent hostname is already known: if a remote agent is not known, the associated tuple is inserted in the local agent map, with the next-hop fields changed to the ones related to the neighbour that sent the list; if a remote agent is already known, the increased distance reported in the considered tuple is compared with the one stored in the tuple already held by the local agent. If the received one is lower, it means that the path associated with the received tuple is shorter and the local agent updates the next-hop and distance fields in its map accordingly.

4. After a local agent updates its agent map, it immediately starts to send the updated map to all of its neighbours, which allows to propagate the information about the network topology across the whole connected component of the graph that represents the MAS.

Figure 4.6 shows the sequence of events that happen during a discovery process in the context of a simple sparse MAS. The results of the discovery process allow the agents to exchange messages in a way that shares some common ground with the concepts behind routing in the IP protocol. By analogy, the agent map held by a local agent would equate to a host routing table. The neighbours of the local agent are similar to hosts on the same subnet that do not require routing to exchange information, and when a remote agent to be contacted is not a neighbour of the local agent, the local agent looks up which is the next hop address, similarly to the *gateway* associated with a certain destination subnet that is reported in the routing table of hosts. Also the agent distance has an equivalent field in the routing table, which is the *metric* value, although with the

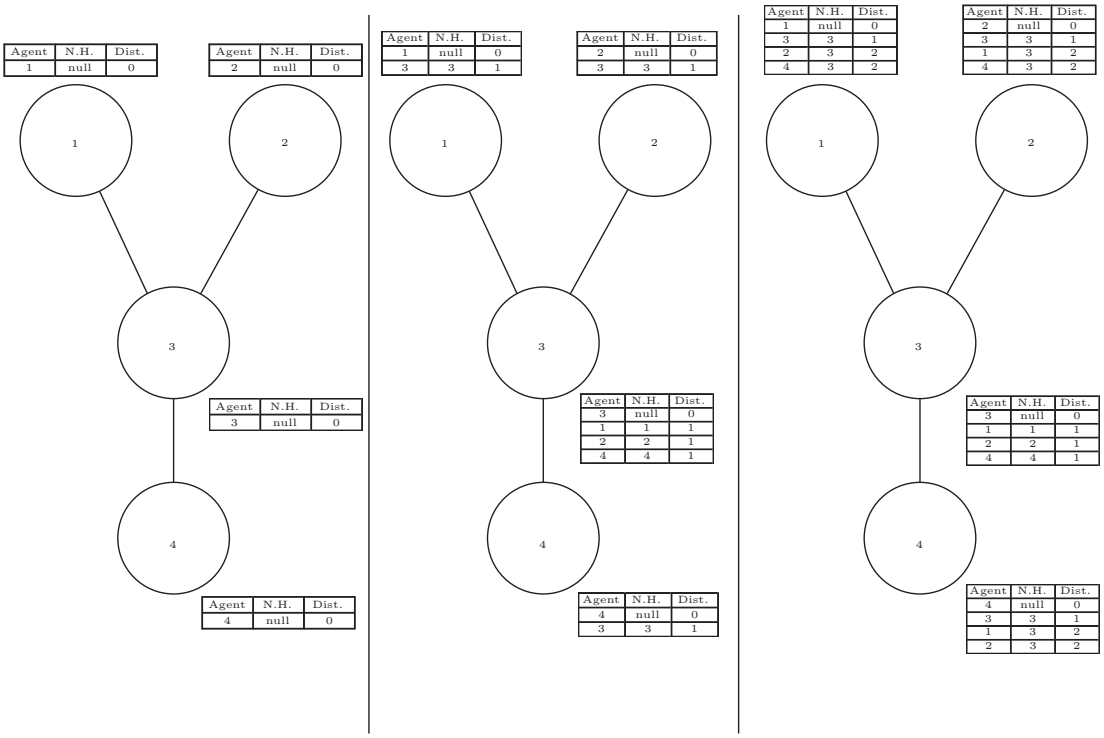

Figure 4.6: Discovery procedure for sparse MASs, where the tables represent the *agents maps*, *N.H.* stands for *Next Hop* and *Dist.* for distance. Initially, every agent puts in its *agents map* only the special tuple about itself (left pane); then, the special tuples of every agents are shared with neighbours, that increase the received distance value by one and set the next-hop reference to the agent that sent the information, resulting in the situation depicted in the central pane; eventually, the updated *agents maps* are shared again through the same process, with agents 1, 2 and 4 notified of the existence of their non-neighbouring agents too (right pane).

current implementation in the MAS case there is only one tuple in the agent map for each remote agent, associated with the information about the shortest path to the remote host and updated when the topology changes. Nevertheless, inserting details about multiple paths towards the same remote agent in the agent map of an agent could increase the fault tolerance of the message exchange as, when the shortest path fails to deliver the message to the remote host, an alternative path to be used would be already known to the sender.

Opinion propagation

The implementation of the consensus algorithm on a MAS represented by a sparse graph requires the introduction of a protocol for managing the propagation of opinions to allow non-neighbouring agents to exchange information. As in a sparse graph there is usually more than one path connecting two non-neighbouring agents, the protocol is needed to select which are the intermediate agents that are required to forward the data.

The protocol that was designed for the purpose is rooted in the concept of **subscriptions**. A subscription is a request that one agent, the *subscriber*, issues to a neighbouring agent, the *provider*: with such a request, the subscriber asks the provider to be forwarded the opinions of some non-neighbouring agent, the latter named the *target* of the subscription. Because of the previously mentioned discovery process that was carried out beforehand, the subscriber is aware that there is a path to the target that traverses the provider; moreover, if many neighbouring agents could serve as provider for the same target, the subscriber will choose the neighbour that ensures the shortest path. Of course, the provider could be a non-neighbour of the target agent as well, in which case it will require a subscription itself to its next-hop agent towards the target agent (if it does not already have a suitable subscription active).

In the code developed for the implementation of the consensus algorithm in sparse MASs, subscriptions are uniquely identified by the tuple *(subscriptor, provider, target)*. In order to implement all the opinion exchanges in the same way, when an agent needs to receive from a neighbour the latter's opinion only, it nevertheless activates a subscription: in this case the provider and the target of the subscription will be equal.

The subscription process is one in which the local agent that needs to receive an opinion of a remote agent plays an active role, requiring other agents to act as forwarders. Other approaches could consist in processes where the local agents play a passive role, receiving the opinions needed because the forwarders detected that they need to. The choice fell on using the former approach as the author believes that latter ones would imply one of the two following conditions:

- The same opinion, with same source and destination agent, is shared in parallel through all the possible paths connecting the source and the destination, because every intermediate agent that detects that the destination agent could require that specific opinion chooses to forward it, regardless of whether that intermediate agent belongs to the shortest path between the two agents. Using this approach would mean that the network traffic is increased significantly, without introducing any relevant benefit.

- Every agent is aware of the whole topology of the MAS, including the edges that do not involve itself, as opposed to be only aware of which are its neighbours and of the next-hop and distance related to the shortest path that connects it to non-neighbouring agents. This would make it possible for an agent to detect if it lies along the shortest path between two agents

– implying that it actually needs to forward the opinions between the two – or if there is a shorter path that connects the two agents, which implies that it should not forward the opinions itself. Anyway, the requirement for an agent to know the whole topology of the MAS would make the discovery process more complex and agents would need to handle a larger amount of data related to the topology, which makes the active-subscriber method more appealing.

As mentioned in Section 3.2.7, one interesting feature of the Self-Focused method is that it allows agents to update their opinion by knowing only the sum of the other agents opinion, without requiring to know separately the single values. In the context of subscriptions, this means that the subscription target can consist of a set of remote agents instead of a single one, requiring the provider to sum the opinions of the agents specified as the target (or to subscribe to another agent for obtaining directly the sum) and send only the sum of the opinions back to the subscriber. With these premises, an additional type of core message must be introduced with respect to the ones mentioned in Section 4.2.1, which is called **Subscription request**, is based on HTTP over TCP port 21852 as for the other core messages and contains the following parameters formatted in JSON:

- **Subscriptor ID**: the hostname of the subscriptor agent;
- **Remote agents ID list**: the list of agents that the subscription is about, whose opinions can be aggregated into their sum;
- **Activate**: a boolean value that specifies whether the involved subscription must be activated or cancelled, as the same type of message can also be used for unsubscriptions when required.

The aggregation of opinions into their sum poses an additional challenge: with the subscription mechanism, agents of the MAS often serve as both subscriptors that retrieve a remote agent opinion through a neighbouring provider and as providers that need to forward opinions to other agents. When an opinion of a remote agent is both received in relation to a subscription to a neighbouring agent and one of those that must be forwarded to another agent, the aggregation with which the opinions are received from a provider could be not suitable for forwarding an opinion to another agent. For instance, if agent n_1 subscribes to agent n_2 for the aggregated opinion of n_2, n_3 and n_4, when agent n_5 subscribes to agent n_1 in order to obtain the opinion of n_2 only (because it could be receiving

the opinions of n_3 and n_4 through another path of the graph), agent n_1 would not be able to satisfy n_5's request because it is not aware of the specific value of the opinion of n_2. The algorithm that was designed to handle these kind of scenarios is shown in the following paragraph.

Management of subscriptions

From the standpoint of a specific agent, the subscriptions in which it is involved can be trivially divided into two classes:

- **Input subscriptions**: those in which the agent is involved as a provider
- **Output subscriptions**: those in which the agent is involved as a subscriber

When the discovery process is complete, an agent immediately determines the output subscriptions needed for executing its update rule: it does so by grouping the elements of the agent map by their next-hop hostname, an action that inserts in the same group all the agents whose opinion can be received by a specific neighbouring provider (including the opinion of the provider itself – this means that even neighbours that are not used as forwarders belong to a one-element group). Then, the agent activates a subscription to each provider, each of which has the associated group of remote agents as target: when a provider acts as a forwarder, sending back to the subscriber not only its own opinion, all the opinions involved are summed as the subscription is just one for all the targets.

Besides the management of the output subscriptions, at any time the agent could need to activate input subscriptions for other agents. When the input subscription request is received, the agent needs to evaluate if with the current output subscriptions it is capable of satisfying the incoming subscription request: it could happen that, by summing its own opinion and the information retrieved through some output subscriptions, the agent is directly capable of providing the requested data to the subscriber. On the other hand, as previously anticipated, it could happen that its current output subscriptions aggregate opinions in such a way that does not allow to identify an opinion subset required by the subscriber. In these cases, the agent detects that it needs to change its output subscriptions to satisfy the new input subscription: it changes them by selecting which agent opinions must be taken out of the aggregation, then it cancels the previous output subscriptions and request others with a lower level of aggregation. An example of such an operation is shown in Figure 4.7. It is worth noting that the designed algorithm takes care of partitioning an existing output subscription in

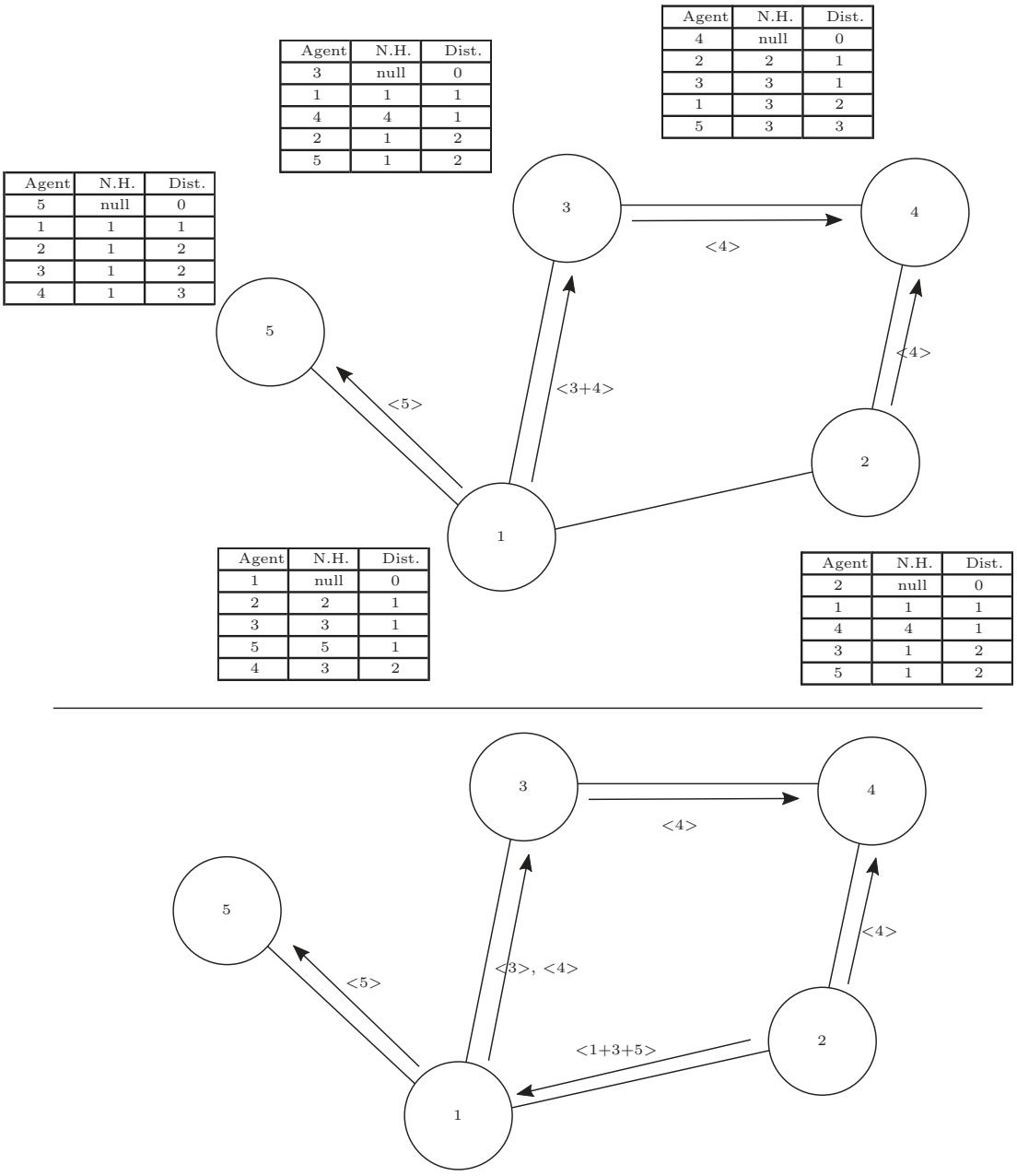

Figure 4.7: An example of input subscription that requires to change the aggregation level of the active output subscriptions. Subscriptions are represented by arrows, with the tail poining to the subscriber, the head pointing to the provider and the target reported as text aside the arrow. In the scenario reported at the top, agent 1 is subscribing to agent 3 for the sum of the opinions of agents 3 and 4; when agent 2 requests a subscription to agent 1 for the sum of the opinions of agents 1, 3 and 5, agent 1 changes its output subscription to obtain the opinions of agents 3 and 4 separately, in order to be able to satisfy the new incoming subscription.

order to satisfy a new input subscription, but it does not guarantee the opposite: for instance, if the new input subscription gets cancelled at a later stage, the partitioned output subscription does not return to be aggregated. An algorithm for the aggregation of separate output subscriptions was considered beyond the

scopes of this project because, in practical terms, it is not trivial to verify that the aggregation of some output subscriptions can be done while keeping all the involved input subscription satisfied. Nevertheless, this aspect should be treated before the implementation in a real-world scenario so that the exchange of messages is always the most efficient possible.

All the techniques described in this Section were then applied in the Docker environment, with the details and the results shown in the following Section and in Appendix A.

4.3 Simulation results

This section shows the results that were produced through the simulations described in the previous section. The results are taken from two sources: the logs printed in the command line interface where the container environment is launched, where the actions performed by the code running inside the containers are traced, and the data stored in the InfluxDB historian, reporting the evolution of the agent opinions in a plot. A separate analysis is reported for the simulation related to the Unpredictable Agent code, which is an implementation of a MAS where all the agents are capable of exchanging direct messages with each other, and for the simulation related to the Sparse Agent code, the one that represents an environment where agents need to exchange messages through multi-hop paths.

Simulation of the Unpredictable Agent

The simulation of the Unpredictable Agent behaviour worked completely as expected. After a debug activity that solved some errors in the developed code, no issue was detected concerning the consensus algorithm or the Docker environment, and it could be verified from the sources mentioned above that the containers were operating properly. As an example of simulation, Figure 4.8 shows a screenshot of the command-line interface used for the simulation, where the top row shows the command used for launching an environment with five agents, then the following ten rows account for the operations that Docker does for launching the environment, after which the messages logged by the single containers are printed, with different colours grouping messages from different containers, a feature that can speed up the analysis of the printed messages. The rows at the center of the screen are related to the discovery process, where each agent becomes aware of the existence of the other agents in the network, while the rows at the bottom

started appearing when the value of 100 for the total amount was set using the coordination dashboard and account for the iterations of the consensus algorithm. Figure 4.9 shows the output for the same simulation taken from the other source,

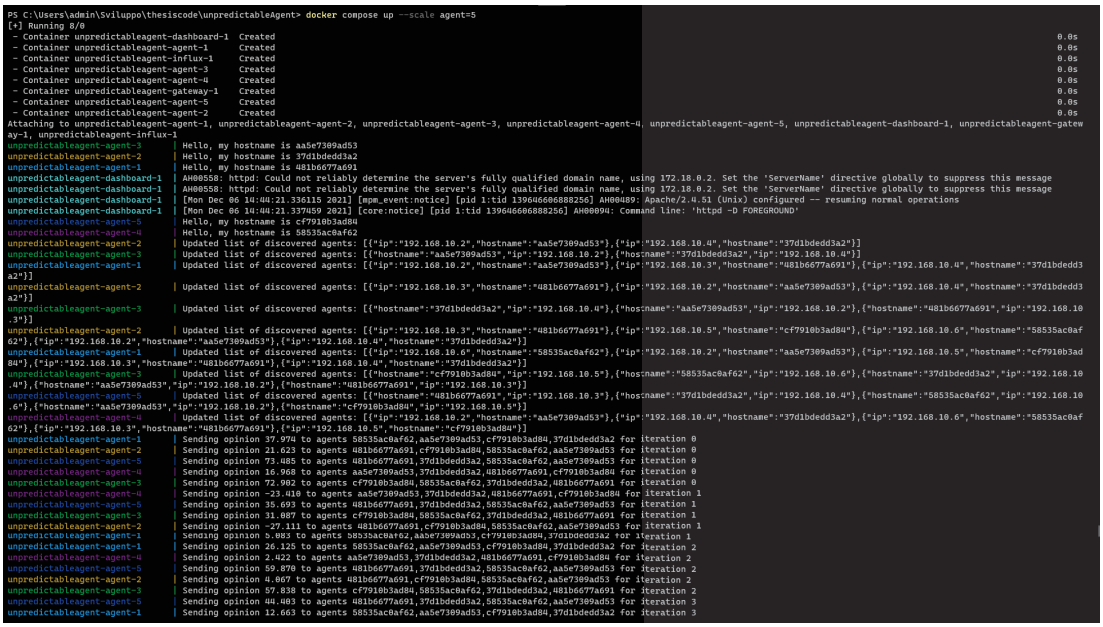

Figure 4.8: A screenshot taken from the command line interface where the set of containers was launched.

which is the InfluxDB webpage, and shows the evolutions of the opinions.

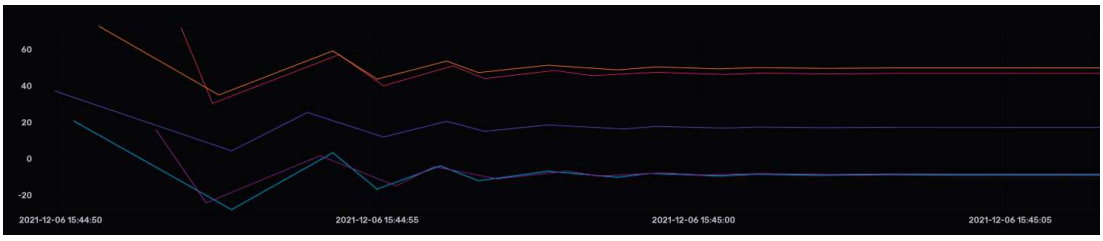

Figure 4.9: A screenshot taken from the InfluxDB dashboard for tracing the results.

Many other similar simulations were run and, when the simulation involved up to 20 agents, no issues were reported. As each of the agent container needs around 175 MB of RAM (a figure that can be retrieved from the Docker Desktop dashboard), when the amount of agents grows further, the containerization environment can misbehave due to an overloading of the host system: as a reference, the author experienced such overloading issues while attempting to run an environment with 30 agents. Nevertheless, in cases where a MAS consisting of a larger number of agents must be simulated, an option could be to use a datacenter with higher resources or a cloud service that allows to run container-based appli-

cations: all the industry-leading cloud services currently allow to deploy Docker container environments onto the user's tenant.

Simulation of the Sparse Agent

For what concerns the Sparse Agent simulation, which is the one related to a MAS represented by a non-complete graph, the generation of the random graph succeeded and the mechanism developed for subscriptions was proven to work properly too. The simulations considered only the *Synchronous one-hop propagation* defined in Section 3.2.7, as it was considered the most relevant because it is the only one in which the dynamics of the overall system are different: nevertheless, in the multiple tests that were carried out, the opinions always converged to a set of quotas that summed up to the total amount requested, although the theoretical definition of the modified system is yet to be addressed.

One thing that was noted is that, comparing two simulations, one related to a complete graph and one related to a non-complete graph using the *Synchronous one-hop propagation*, using the same number of agents and the same initial opinion and weight chosen by each of the agents, the quotas eventually assigned to each agent are different between the two simulations. By denoting the distance between two agents in the graph as d_{ij}, and by defining the maximum distance of a agent i as

$$d_{i,MAX} = \max_j d_{i,j}$$

it was found that agents that have higher value for $d_{i,MAX}$ – which are the most peripheral in the graph – are favoured in obtaining a final quota closer to the initial opinion they chose. This is explained by the fact that, with the implemented algorithm, agents begin to trade-off their initial opinion with the opinions expressed by other agents only after they received the initial opinions of all the other agents, and with the *Synchronous one-hop propagation* this happens after $d_{i,MAX}$ iterations; before that moment, they simply repeat their initial opinion in subsequent iterations. Agents with a lower value for $d_{i,MAX}$ begin to drift away from their preferred opinion earlier in the iterations, therefore the quota they are eventually assigned is farther from their initial opinion. As a simple example, consider the same scenario of Figure 3.1, with three agents in a line connected in pairs: with respect to the associated completed graph (that has an additional edge between agent 1 and agent 3), agent 2 is penalised in the consensus algorithm, as it begins to compute the weighted average with the other agents' opinion at the first iteration, while agents 1 and 3 wait an additional iteration before doing so.

Figure 4.10 show the comparison between the two cases.

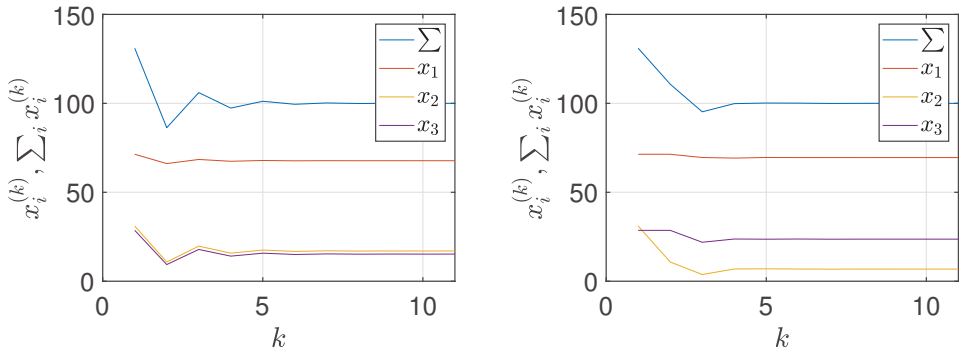

Figure 4.10: Comparison between the evolution of opinions in a complete graph (left) and in a non-complete graph using *Synchronous one-hop propagation* (right), referring to the graph of Figure 3.1, with a Total amount of 100 and same initial opinion and weight per agent. For the non-complete graph case, it can be noted that opinions of agents 1 and 3 do not change at the first iteration, and these agents obtain a final quota closer to their initial opinions compared to the same quantity in the complete graph case.

In the rest of this section, two examples of randomly-generated Multi-Agent System upon which the *Synchronous one-hop propagation* was run are described. The parameters used in the simulations are reported in the following:

- Number of agents: 5
- Number of multicast groups available: 4
- Bernoulli parameter: 0.4
- Total amount: 100

The random graph generation resulted in the graphs shown in Fig. 4.11, while the evolution of the opinions in the two simulations is reported in Fig. 4.12. Figure 4.13 reports the log related to the launch of the container environment during simulation A, where the information about the discovery process and the subscriptions is reported. The updates about the agents map are in JSON format, while the subscriptions that are activated are mentioned in the form *subscriptor->provider->target*. In the last five rows of the log, two subscriptions to aggregated targets are shown, which are then satisfied when the iterations of the consensus algorithm start as shown in Figure 4.14.

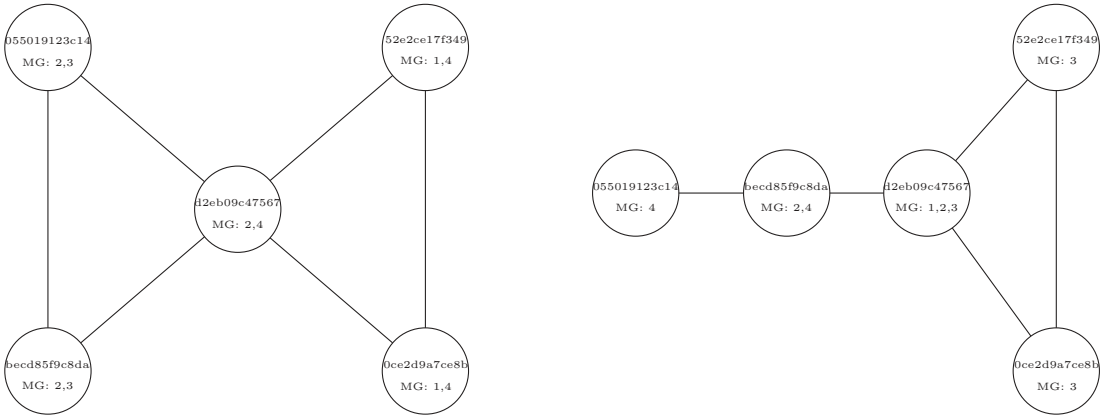

Figure 4.11: Sparse graphs generated: on the left, the graph of simulation A; on the right, the one of simulation B. The text within the circle representing each of the nodes reports the agent hostname and the multicast groups that the node joined. Agents that share at least one common multicast group are neighbours.

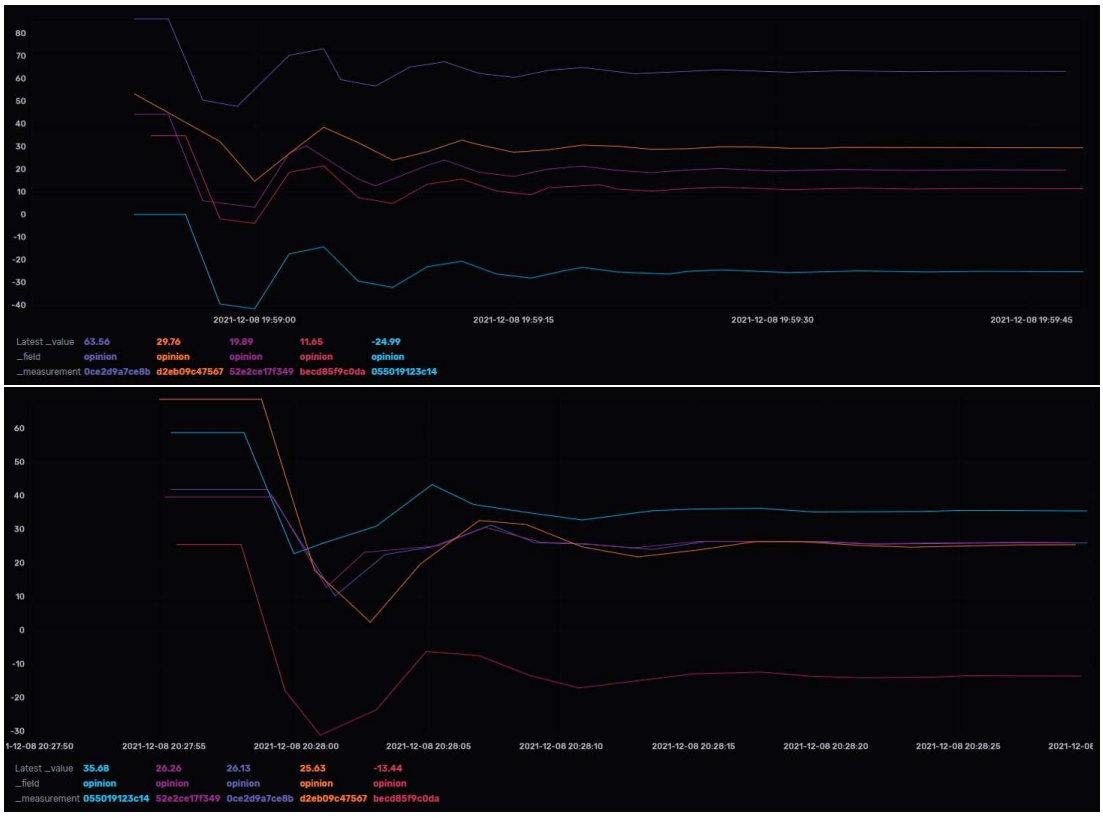

Figure 4.12: A screenshot of the opinion evolutions in simulation A (top) and B (bottom).

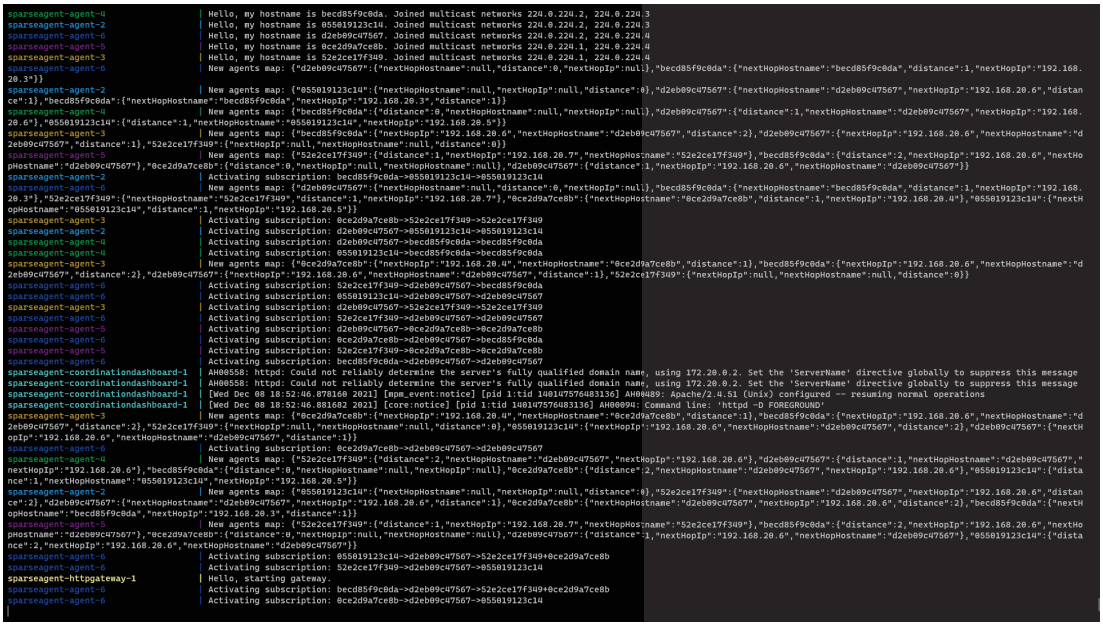

Figure 4.13: A screenshot of the log produced by the environment at its start during simulation A.

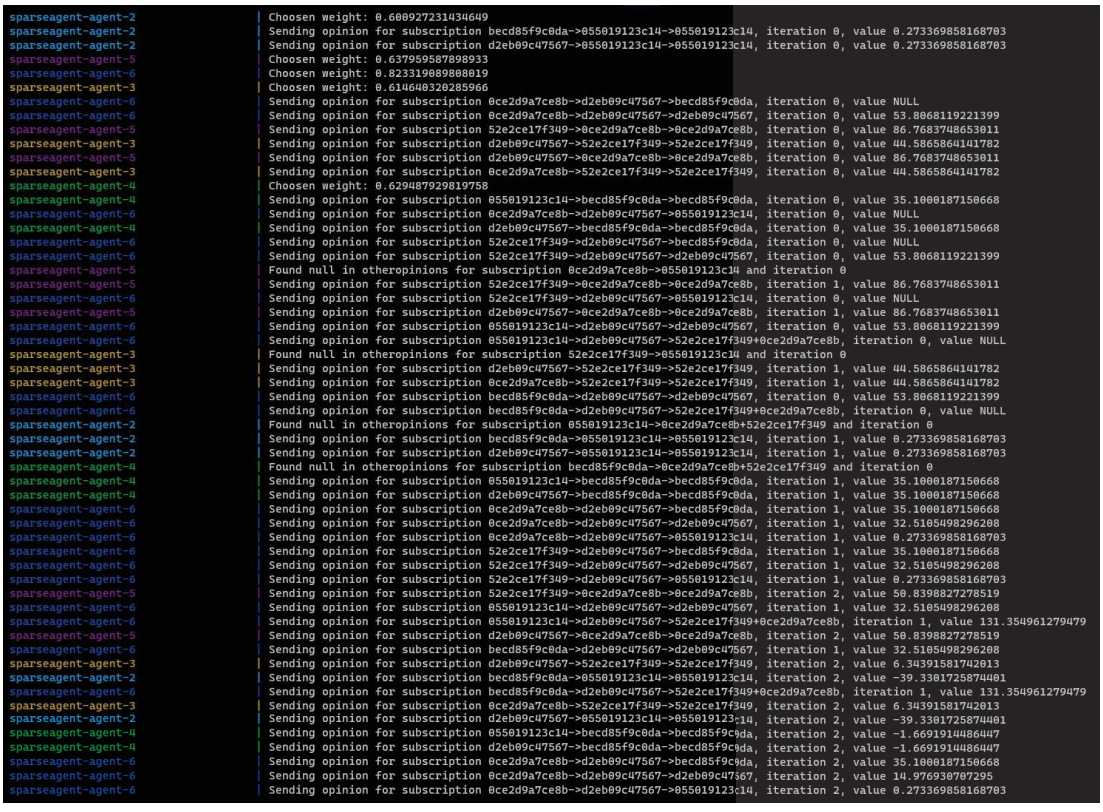

Figure 4.14: A screenshot of the log produced by the agents when the iterations of the consensus algorithm for simulation A begins, showing the propagation of two subscriptions related to the aggregated opinions (the ones in which the target consists of two agent hostnames concatenated with a plus sign).

4.4 Activities to implement the code in a real-world scenario

The following list includes the activities that the author believes are needed before an implementation on a specific real-world scenario can be done:

- For a specific use case, the definition of the optimization logic related to the initial opinions, the weights, and hard limits is required. As the study is focused on the design of a general framework for the allocation of quotas, this part was not addressed in the simulations: the choice of initial opinions and weights is done randomly and the hard limits are not even implemented. However, given a certain process to be controlled, it should not be hard to define the optimal quota that an agent aims to be assigned (the initial opinion), how much it is prone to be assigned a different quota than its chosen one (resulting in the weight choice) and which are the boundaries on the quota that it can take care of (the hard limits).

- As the initial opinion choice is meant to be based on some local variables under the single agent's domain of control, and the quota that an agent is assigned at the end of the consensus procedure must be actuated in the controlled plant, in a real-world scenario the software needs to exchange I/O signals with sensors and actuators. This is a design phase that is specific to both the hardware where the software runs and the specific application which the algorithm is applied to, therefore it was not considered during simulations. Moreover, an agreement with the external party that sends the total amount value must be reached about the protocol to be used for this purpose, not necessarily being the JSON payload over an HTTP connection as it was done during the simulations.

- More generally, the protocols used in the simulation environment could not represent the best choice, especially when considering industrial environments. The replacement of HTTP with some industrial protocols such as Modbus TCP or IEC 60870-5-104 could represent some more efficient and robust alternatives to exchange the same payload, while keeping the physical media and the structure of the communication almost unchanged. They were not used in the simulation because this would have increased the complexity of the code without relevant benefits.

- As it was developed for the simulation, the code is completely lacking of cybersecurity measures that must be adopted in a real-world scenario. In fact, for a malicious agent that aims to alter the result of the consensus process to its convenience, the only thing that is needed is the physical access to the network that the agents are connected to. This issue can be mitigated by acting at the data link or network level without changing anything in the agent code, implementing techniques such as VLANs or IPsec tunnels with a configuration that ensures that only the authorized agents are allowed to join the network. Nevertheless, a far better approach would be to implement cybersecurity measures at the application level: first of all switching from HTTP to HTTPS as the application protocol used for exchanging data between the members of the network (provided that industrial protocols are not used in place of HTTP), using TLS certificates signed by an authority trusted by all the agents, so that confidentiality and server authentication are implemented – where the server in this case is the agent receiving a message; for client authentication – the authentication of the agent sending the message – one could either opt for client TLS certificates implementing the same approach of the server side, or some type of authentication through credentials or tokens could be implemented. Moreover, if InfluxDB is used for as the historian for the agent's data, it must be protected too by using HTTPS and a more secure authentication method, both for user access and the access of agents that connects to InfluxDB for recording their data.

Chapter 5

Conclusions and further developments

The author believes that the study carried out for this thesis yielded multiple results. The main result is arguably the theoretical definition of the two distributed solutions to the problem of allocating quotas, the Exact-Breakdown and the Self-Focused methods, where the first was found to be very similar to one already discovered in a different context – finding consensus on the opinions expressed by many scientists on a stochastic distribution, while for the second there is no similar study known to the author. In fact, the problem of allocating quotas through a distributed procedure itself was found to have few references in the literature, making the study relevant as fields where such a solution could be applied are many. As reported in the introduction, one of the fields of interest for the application of these solutions is the energy sector, that in the following decades will need to transition further from a centric paradigm where the most relevant share of power generation happens on few, large plants, to a paradigm where the power generators are many and smaller. And if the plant to be controlled has a distributed structure, a distributed control system attracts more interest because it fosters redundancy and fault tolerance.

Besides the energy sector, the results represent a general purpose approach that can be applied to a wide range of fields. The stability conditions of both theoretical methods were addressed, making them applicable to a real network of agents without requiring further studies on the theory, at least for what concerns the case in which all agents are allowed to exchange messages directly with each other. The Self-Focused method seems to be promising also for applications on Multi-Agent Systems that are represented by a non-complete graph, where some

of the agents must serve as a gateway for exchanging information between two other agents, which was empirically tested but further studies are required to apply the method in these scenarios. However, if the assumption that opinions propagation can happen within the time window between two consecutive iterations of the consensus algorithm hold, the Self-Focused method can be directly applied in sparse scenarios as well, as the dynamics of the consensus algorithm would be exactly the same of those that characterise the scenario where all the agents can communicate with each other, but this is an assumption that would hardly hold when the number of agents in the MAS and the diameter of the graph representing the MAS grow.

Another relevant result, which came as a sort of side-effect of the study, was the usage of Docker containers for the simulation of a generic Multi-Agent System, which probably represents an unusual reason to use containers but was proven to be effective: it allowed to define and test the developed code in an environment hosted on a single device, while keeping it close to a real-world environment, where multiple instance of an agent type run on different devices connected to the same network. Indeed, all the communication protocols that define the data exchange among agents and between an agent and the external network were implemented and tested, making them ready for the deployment on a real-world environment. The author believes that this way to simulate the operation and the interaction of the software running on the agent hardware could be reused in many other MAS-related projects, as it speeds up the developing and testing activities. In order to encourage the usage of such a simulation platform, as mentioned in the related Chapter 4, the code base developed is published on a free access Git repository, which includes seventeen *snippets* that represent simple containerized programs that perform basic tasks. They were developed as the first steps that the author himself did while learning how to develop in the Docker environment, and they are included in the repository to allow for a quick learn of the basics behind the topic, without the need to use as a reference the more complex code developed for the actual simulation. Although almost all of the code published in the repository is written in PowerShell language, the author believes that these snippets could be quickly ported in any other containerizable language that a developer could be familiar with.

This study leaves many challenges open for further developments, the most important being the theoretical study of the consensus algorithm when using *Synchronous one-hop propagation* in sparse Multi-Agent Systems (see Section 3.2.7):

if the validity of the method is confirmed, this distributed approach for allocating quotas would be useful, especially in large and sparse networks, where the usage of the *Multi-hop propagation* could slow down the consensus algorithm excessively. Another aspect that was not addressed about the sparse MAS scenario is how to reaggregate the output subscriptions on the basis of the changed input subscription (see Section 4.2.3), a study that should not be really complicated and would optimize further the data exchange on a sparse MAS. On the other hand, for what concerns the application to MASs represented by complete graphs, a yet to be addressed topic is the evaluation of the convergence time of the Exact-Breakdown method and the related constraints to be imposed when selecting an agent's weight.

It should be noted that all the methods defined in the theoretical part assume that, before the consensus algorithm iterations take place, each agent chooses its initial condition and weight (consisting of an array in the Exact-Breakdown case and of a scalar value in the Self-Focused case). Although in the project for the course *Networked Control for Multi-Agent systems*, where the basis of this thesis are rooted, the problem of defining how to choose the initial condition and weight was addressed according to the specific application's characteristics, throughout this thesis such a problem is never addressed because it is specific to the application that the method must be applied to, while the thesis is meant to provide a general framework. During the simulations such a choice was made at random, but of course whenever this method needs to be applied to a real-world scenario the choice logic must be defined. Anyway, for each application it should be easy to define a logic that expresses an agent's preference on the quota to be assigned and its availability to drift away from its preference, expressed in the form of a lower or higher weight. Such a choice is meant to be based on an optimization process computed locally, by considering the state of the plant governed by the single agent.

The last theoretical open point is related to how the agents are rewarded for their flexibility: in applications where the agents of a MAS are controlled by different non-cooperating entities, each of the agents could be tempted to raise its weight (or self-weight in the Exact-Breakdown case), aiming to yield a quota as close as possible to its optimized candidate quota expressed through the initial conditions. If many agents raise their weight, the consensus process will slow down; moreover, as the impact of weights is relative, a hostile agent could try to raise it more and more until its weight is significantly higher than the

other agents', possibly leading to a chain reaction that could bring many weights very close to one, making the consensus-seeking process unacceptably slow. In applications where the specific activity that the agents execute together results in an economic reward that must be divided among the agent (power generation could again be one example), a solution could be to implement a business model that rewards each agent basing upon the combination of two factors: the quota they where assigned through the consensus process – which accounts for the actual production they contributed to – and the weight or self-weight that they chose. The latter factor should be considered in a way such that the revenue of agents that specified a higher weight for their initial opinion is reduced. Using this method, the flexibility offered by agents that specify a lower weight is rewarded, and it is expected that agents themselves are led to identify a trade-off between the willing to obtain an optimal quota and the revenue they plan to gain. Especially in the Self-Focused case, agents are able to determine which weight was chosen by each of the other agents by inverting their update rule, therefore the application of the rewarding scheme could be distributed as well, without the need for an external central authority that computes and assigns the revenue quotas. The definition of such a business model was not addressed in this study, and the author believes it represents an interesting evolution.

Appendix A

Container code descriptions

In this section, further details about the developed software for each of the containers are reported, including flowcharts that describe the high-level structure of the code and details about configurations.

Gateway

The gateway container implements a single point of contact between the user and all the agents. It takes care of receiving requests and commands from the user – who triggers them thanks to the Javascript code included in the Coordination dashboard described in the following Section, forward commands to agents upon need and return replies to the user. With reference to message types listed in Section 4.2.1, it handles:

- The reception of broadcast *discovery messages* from agents, in order to update a local known agents table that is needed for handling the message types reported in the following.
- The reception of the *Total amount* command from the user, which reports the overall amount to be split into quotas, forwarding it to all the known agents. The gateway also takes care of storing the updated *Total amount* value in the InfluxDB database to allow for analyses.
- The reception of the *Get Agents list* command from the user, replying with the list of currently discovered agents, which is then shown in the Coordination dashboard.
- The reception of the *Exit* command from the user, that is forwarded from the gateway to all the agents before triggering its own exit process.

An additional feature that was developed in the gateway code in order to allow the user interaction was the configuration of the CORS-related HTTP headers. CORS, an acronym for *Cross Origin Resource Sharing*, is a technique that allows an external HTTP resource to be accessed using some code that was retrieved from a page hosted on a different domain, an action that would normally be blocked by the browsers due to security reasons. In the case of this project, the resource is the gateway API and the page is the one provided by the Coordination dashboard. In order to allow the action, the external HTTP resource must explicitly specify that it consents being accessed from other webpages, and it does so providing the client with the additional HTTP headers *Access-Control-Allow-Headers*, *Access-Control-Allow-Methods*, *Access-Control-Max-Age* and *Access-Control-Allow-Origin*, each one with properly configured values. In particular, the last of the four headers must report the domain of the webpages that are allowed to make the client contact the external HTTP resource: for the sake of simulation only, this header was set to the wildcard value "*", which practically means that any webpage can make the user contact the gateway. For real-world applications, a more restricted policy must be implemented to improve the security level of the whole system.

Coordination Dashboard

The Coordination dashboard is a basic web application. It is implemented starting from an Apache webserver official container, adding an HTML page and a Javascript file containing the functions needed for the interaction with the gateway. The webpage is made of three sections, each of which has a related Javascript function:

- A section to send a value for the *Total amount* command: when the user hits the associated *Send* button, the *sendTotalAmount()* Javascript function is triggered, which sends the value inserted in the corresponding textbox as a *Total amount* message to the gateway.

- A section to retrieve the agent list: when the associated *Get* button is clicked, the *getAgents()* Javascript function is triggered, which sends a *Get Agent List* message to the gateway, then parses the response payload and shows the agent list below the button.

- A section to launch the *Exit* command: when the associated button is clicked, the command is sent to the gateway through the *stopAll()* Javascript

function.

The appearance of the Coordination Dashboard webpage is presented in Figure A.1.

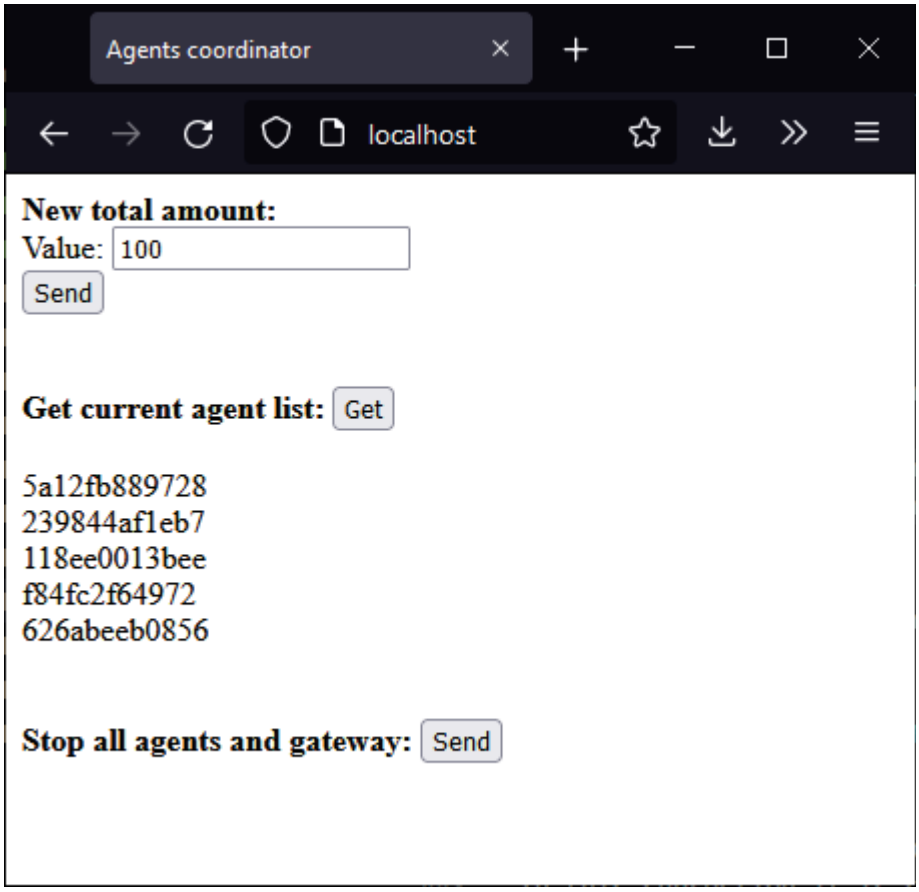

Figure A.1: Appearance of the Coordination Dashboard, related to a simulation of a MAS consisting of five agents. The five alphanumeric codes reported in the center of the page appeared when the *Get* button was clicked, and are the hostnames of the five Docker containers representing the agents.

InfluxDB

The container related to InfluxDB was used as it is published on the Docker Hub, only adding some custom configuration related to the following settings. In fact, if the InfluxDB container image is launched straightforwardly, at the first connection from a browser a configuration wizard is shown. Through this wizard, the user is required to set the following parameters about the instance:

- The name of the *Organization* that owns the instance
- The name of the initial *Bucket*, where a Bucket defines a subset of the data stored in the InfluxDB instance – many of these can be deployed if there is

the need to store totally different datasets on the same InfluxDB instance, which is not the case of this project

- The username and password representing the initial credentials to access the InfluxDB instance webpage, through which the initial user can create more logins at a later stage

At least for what concerns the simulation environment, custom settings were not needed, therefore a standard configuration was applied through *environment variables* of the container as reported in the InfluxDB Docker image official webpage[1]. With this procedure, the InfluxDB container instance can be used directly after being cloned from the image, without requiring an user to go through the wizard before being able to use the instance. Besides the parameters that the user would need to insert in the wizard, three additional parameters were preset:

- The *initial token*, another kind of credential meant to be used to authenticate an application – in this case the code running on the other containers – instead of a user

- The log level, which was set to *Error* instead of the default value of *Info*: if the Docker compose command is used to launch a set of containers as was done for the simulations, all the logs from the various containers are shown in a single window, which in our case includes both the logs from the agents and those of the service containers. Allowing the InfluxDB to display information logs makes it harder for the user to understand the behaviour of the agents during the simulation, because their logs are obfuscated by the InfluxDB ones

- The retention time, which is the time interval after which the data is deleted from the DB. The interval was set to one hour, as for the simulation the InfluxDB is meant to show only the data about the last simulation runs. Without this setting, choosing the data to be displayed for the simulation analysis would be harder, as all the past simulation data would be listed among the most recent simulation data

Of course, these kind of presets are not meant to be used in real-world scenarios, especially for what concerns the security aspects of having standard hardcoded

[1] See `https://hub.docker.com/_/influxdb/`, especially the *Using this Image - InfluxDB 2.x* section.

credentials valid for all the implementations. Nevertheless, using a different configuration, InfluxDB is deemed an easy and valuable way for storing the agents' data also in real-world implementations.

After the entire simulation environment is launched (but the operation in a real-world scenario would be almost identical), InfluxDB can be accessed through a web browser exposed on the default port TCP/8086. Users can log in using the default credentials configured for the simulation, access the *Explore* section of the application, filter the data they want to display – possibly all the recorded data – and analyse the simulation results. Figure A.2 shows an example of InfluxDB data that can be accessed through a browser.

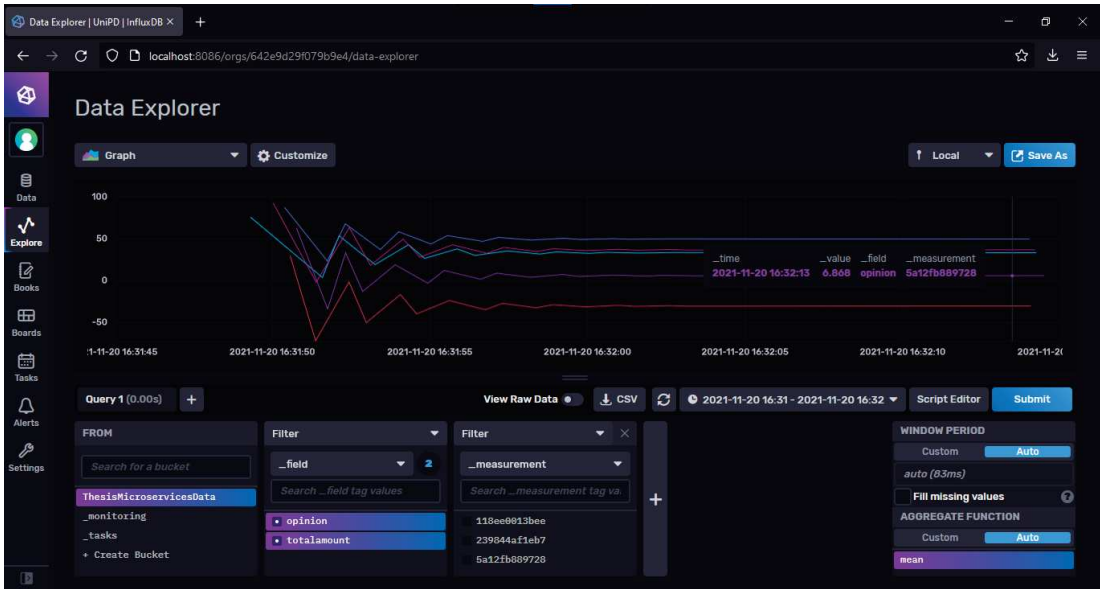

Figure A.2: Example of how InfluxDB data can be accessed through a web browser

Unpredictable agent

The *unpredictable agent* is the most basic implementation of an agent logic, made for the only purpose of testing the consensus algorithm, the communications among the agents and the overall container-based environment. The code structure described in Algorithm 1 was developed and, as its name aims to suggest, the choice of the initial opinion and weight is made at random. More specifically, the initial opinion choice is made extracting a random value from a uniform distribution ranging between 0 and the total amount requested by the user; the weight is extracted from an exponential distribution, which support is properly scaled in a way such that:

Algorithm 2 Algorithm used for weight selection in the *Unpredictable agent*. Variable o_p represents the desired probability of having an outcome higher than one, used for scaling the exponential distribution parameter; λ is the resulting parameter for the exponential distribution; $U[0,1]$ represents the extraction of a uniform random variable in the range $[0,1]$; w_{MIN} is the minimum weight determined from the results of Section 3.2.3; w_n is the exponential variable extraction normalized in the range $[0,1]$.

$o_p \leftarrow 0.001$
$\lambda \leftarrow -\log o_p$
if oscillatory behaviour allowed **then**
    $w_{MIN} \leftarrow \frac{n-2}{n}$
else
    $w_{MIN} \leftarrow \frac{n-1}{n}$
end if
$w_u \leftarrow U[0,1]$
$w_n \leftarrow -\frac{1}{\lambda}\log(1-w_u)$
if $w_n > 1$ **then**
    $w_n \leftarrow 1$
end if
$w \leftarrow w_{MIN} + w_n * (1 - w_{MIN})$
return w

- the minimum value possible corresponds to the lowest weight that ensures the stability of the consensus process according to the outcomes of Section 3.2.3 (which means that the lowest weight depends upon the number of agents in the MAS). The code allows to quickly select whether the oscillatory behaviour is acceptable or not by changing the value of a boolean variable and setting the minimum value accordingly.

- as the exponential distribution function is always decreasing but never reaches 0, it is scaled so that the probability of extracting a value above 1 is equal to 0.1%; after the extraction is performed, if the extracted value exceeds 1 it is nevertheless forced to 1.

The process that performs the weight choice is detailed in Algorithm 2, where the *Inversion method* for extracting a sample of an exponential distribution starting from a uniform distribution was used. Further details on this consolidated method can be found for instance in [8]. The extraction of the random weight was performed in the mentioned way, without opting for a uniform random variable as was the case for the initial opinion, to simulate the fact that agents should choose weights that are as low as possible to speed up the consensus process, but con-

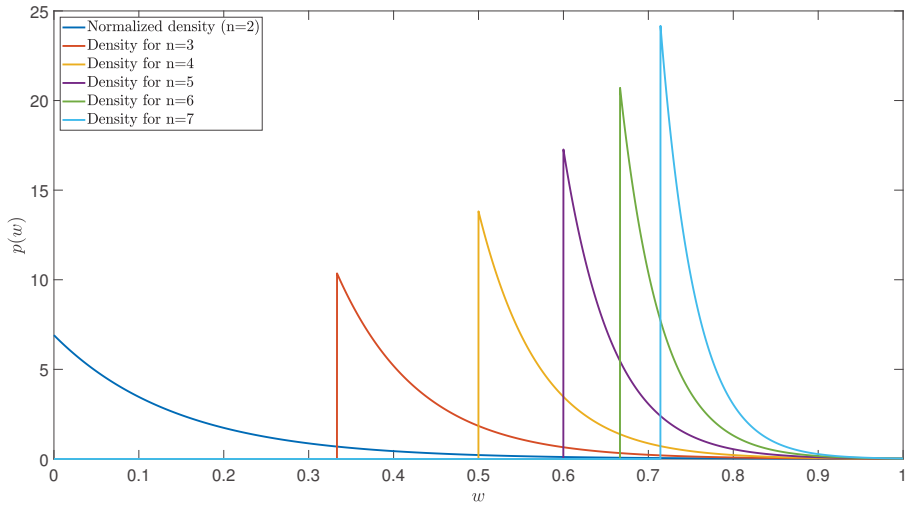

Figure A.3: The probability density functions that represents the distribution used for the weight choice as the number of agents change, corresponding to the setting of allowing oscillatory evolution for the opinions. If the oscillatory behaviour is not allowed, the corresponding densities would be pushed farther to the right due to the increased w_{MIN} factor.

sidering that agents can raise their weight in critical cases when they need their assigned quota to be closer to their initial opinion. The distribution that was simulated with the algorithm is represented in Figure A.3, where the dependance on the number of agents is highlighted.

In order to allow for an easy change of the initial opinion and weight choice logic, the PowerShell modules that implement these processes were kept separated from the rest of the code, so that different logics can be quickly implemented by changing these modules only. Eventually, Figure A.4 shows the complete flowchart that represents the code implemented for the Unpredictable Agent.

Sparse agent

In order to test the implementation of the logic to be used when the graph representing the MAS is not complete, as a result of the considerations expressed in Sections 3.2.7 and 4.2.3, the code related to the Unpredictable agent described in the previous paragraph was changed. Only the *Synchronous one-hop propagation* approach was considered in this test. The parts that were added or changed with respect to the Unpredictable Agent code were the following:

- The choice of the multicast groups to join for the sparse graph generation (see Section 4.2.3). The number of available multicast groups was set to six as default value, but it can be easily changed by setting an environment

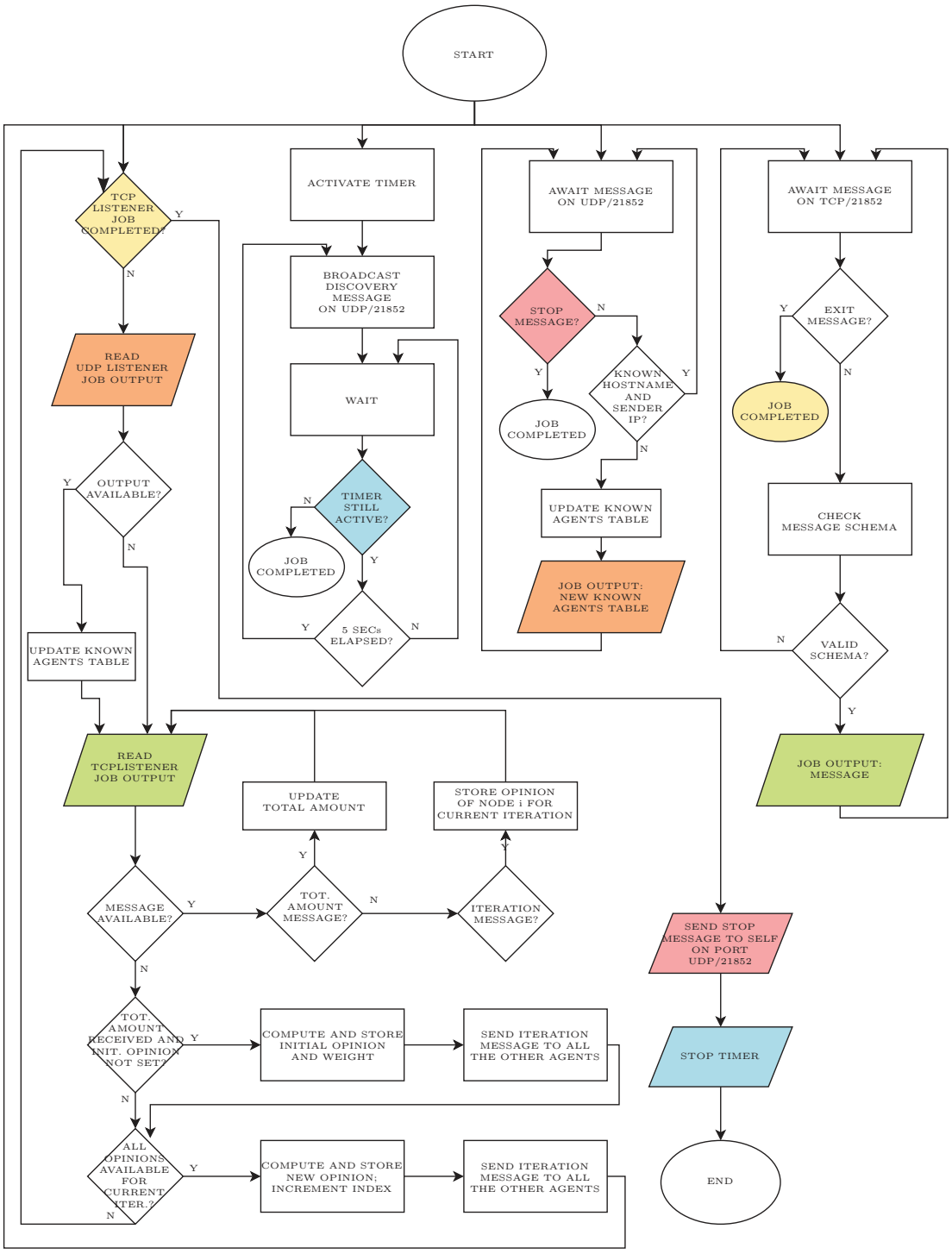

Figure A.4: Flowchart representing the *Unpredictable agent* code. When the program starts, four parallel processes are launched, represented in the flowchart by four branches: the leftmost branch represents the main process of the program; the second branch represents the process that periodically sends a broadcast discovery message to all the other agents; the rightmost two branches represent the processes that handle the synchronous reception of UDP and TCP messages and pass their outputs to the main process. Input-output relations among the four processes are highlighted using shapes with the same background colour.

variable named `CHANNELNO` in the command line interpreter before launching the containers through the `docker compose` command: the reason why the parameter can be set this way lies in the fact that also the gateway container must be aware of the number of available multicast groups, as explained later in this section, so the usage of an environment variable allows to specify the parameter value only once and make both container types aware of it. On the other hand, a parameter that does not need to be known by more than one container type is the Bernoulli parameter, which defines the probability that an agent joins a multicast group, and is hardcoded in the sparse agent code. The default value of 0.4, but it can be changed directly in the code. The extraction of an outcome from a Bernoulli random variable is performed by extracting the outcome of a uniform random variable in the range $[0, 1]$ and comparing the outcome to the Bernoulli parameter: if the outcome of the uniform random variable is lower, the outcome of the Bernoulli random variable is set to 1, otherwise it is set to 0.

- The management of discovery messages, both the ones to be sent and the reception of those sent by other agents. As explained in Section 4.2.3, in the case of non-complete graphs the discovery messages cannot contain only the hostname of the sending agent, they need to contain the data structure called *agent map*. The agent map itself is a data structure not present in the Unpredictable agent that is handled by the sparse agent code, and replaces the *known agents* data structure of the Unpredictable agent that contained the association between the hostname and the IP of all the other agents of the network (which in that case were all neighbours).

- The management of subscriptions: a new JSON schema was defined to represent how the *Subscription request* message should be structured, and the logic for sending the subscription requests needed to receive all the agents opinions was developed. Moreover, the code needed for handling incoming subscription requests was developed, implementing the logic described in the paragraph 4.2.3, which updates the output subscriptions by dividing a subscription to aggregated targets into many more granular subscriptions, when the current output subscriptions would not allow to serve a new input subscription request. In order to streamline the algorithm that manages the input and output subscriptions, the opinions needed by the agent

to compute its own update rule were considered using an additional input subscription associated with the hostname of the agent itself, as if the agent was requiring the involved opinions to itself.

- The aggregation of received opinions that must be performed to satisfy an input subscription: this was the most challenging part to implement. Maintaining a data structure with links between each input subscription and the output subscriptions needed to satisfy the former was deemed a too complex process, therefore a different approach was used. Whenever an agent needs to evaluate the value to be sent to a subscriber, it loads the set of agents that represent the target of the subscription, removes from the set the reference to the agent itself if present, then calls a function that computes all the possible partitions of the set: for instance, if the input subscription has the agent set $\{a, b, c\}$ as target, the function returns the partitions $\{\{a\}, \{b\}, \{c\}\}$, $\{\{a, b\}, \{c\}\}$, $\{\{a\}, \{b, c\}\}$, $\{\{a, c\}, \{b\}\}$ and $\{\{a, b, c\}\}$. Then, the algorithm checks whether any of the returned partitions is composed by sets that are all existing output subscriptions: the existence of one matching set of output subscriptions should be guaranteed by the management of input and output subscriptions described above. When the set of output subscriptions that are needed to satisfy the input subscription is detected, values received from the involved output subscriptions at the previous iteration are summed, the own opinion of the agent is summed as well if the agent itself was included in the target of the input subscription, and the result of the sum is sent to the subscriber.

  As the function that computes all the partitions of a set, which code was retrieved online (see Appendix B), was hard to be translated into PowerShell due to the data structures it relied on, an additional feature of PowerShell was used: by means of the `Add-Type` command, PowerShell is capable of loading .NET classes either compiled in DLL files or by compiling at runtime some C# language code. The latter option was used: the specific algorithm was used as it was found online, written in C# language, and loaded into the PowerShell script through the `Add-Type` command.

  It is worth to stress that the implemented algorithm uses the *synchronous one-hop propagation* method, meaning that at every iteration of the consensus algorithm an opinion can proceed of one hop only towards a remote agent. This means that an agent, while managing correctly its output sub-

scriptions to satisfy an input subscription, could not have received a target opinion itself, especially during the first iterations of the consensus algorithm. These cases are handled by using the placeholder value *null* to describe such situations and signal to the subscriber that the requested opinion is not yet available; when an agent receives at least one *null* value from its providers, it is not capable of performing the update rule, therefore at the next iteration it will stick to the same opinion value sent during the previous iteration. As mentioned in Section 3.2.7, this introduces a delay whose effects need to be evaluated further.

- The last thing that was slightly changed was the *Iteration* JSON schema related to the exchange of opinion among agents: instead of reporting the single hostname that the transmitted opinion is about, it reports an array of hostnames in order to account for the possibility of transmitting aggregated opinions.

The simulation was then run by replacing the Unpredictable agent container with the Sparse agent one and using all the other containers almost unchanged, with the sole exception of the Gateway that was slightly modified so that it could receive discovery messages sent to all the multicast addresses associated with the available groups, which amount is retrieved from the `CHANNELNO` environment variable or set to the default value of six if the environment variable is not set. In fact, the gateway container needs to be aware of the existance of all the agents, regardless of their role in the topology, so that it is capable of sending them the total amount value received from the user and can reply correctly to the user's request of listing the available agents on the coordination dashboard.

The case in which an agent does not join any of the multicast groups was not handled: in such a case, the other agents of the MAS will not become aware of its existence, therefore they will not share their opinions with it; the gateway will not be aware of the isolated agent either, therefore the latter will not receive the total amount value sent by the user. This means that the isolated agent will not take part in the quotas allocation process, and the author believes that this behaviour correctly accounts for real-world cases in which one agent would not be able to communicate at all with the MAS it belongs to. Another possible scenario could make the randomly-generated graph be composed by two or more connected components: in this case all the connected components would try to reach consensus on the allocation of quotas of the total amount, independently from the other connected components, as if the latter did not exist. This be-

haviour is undesired, but a solution to overcome this issue was considered beyond the scopes of this study.

Appendix B

List of software used in the project

The following list includes the software and tools that were used throughout the study along with the reason why they were chosen:

- **Visual Studio code**: a free source code editor published by Microsoft, which allows to write and test code in many languages, all in a single environment. It was used for both writing this document in LaTeX and for the development of the code to be containerized in PowerShell, HTML and JavaScript; moreover, it was used to write the *dockerfiles* and the *compose files* needed for setting up the containerized environment. Visual Studio Code allows to install *extensions* that can be downloaded from the Visual Studio marketplace, which are useful when developing in a particular programming language as they provide syntax highlighting for the language and integration with the interpreter of the language. For this reason, the following extensions were installed and used, most of which integrate with other standalone software reported below:
  - LaTeX Workshop
  - PowerShell
  - Docker
  - PlantUML
- **TeX Live 2021**: a LaTeX environment that includes fonts, LaTeX packages and a compiler, among other tools. Its integration with Visual Studio Code was used for editing this document.
- A LaTeX template made for thesis of the Department of Information Engineering of the University of Padova, kindly shared on GitHub by the user

@mamio1994[1].

- **PowerShell v7.2** (see Section 4.2.2)
- **PlantUML**: a free software based on Java that renders diagrams defined by the user through text files. It was used to create the UML sequence and network diagrams.
- **Docker Desktop**: a package that contains the Docker engine and some tools that are useful to manage containers (both graphical and command-line).
- **Windows Subsystem for Linux 2 (WSL)**: a component that allows to run executables made for Linux on a Windows machine, with version 2 implementing a real Linux Kernel. Its installation is a requirement for running the Docker engine on Windows.
- **Git**: a source code versioning tool that was used both while writing this document and for the source code developed for the containers. The online repository that was used is the one of the Department of Information Engineering of University of Padova, hosted at `https://gitlab.dei.unipd.it`.
- A C# algorithm used to find all the possible partitions of a set, uploaded on StackOverflow by the user Daniel Wolf[2]. See the part of Appendix A related to the sparse MAS simulation.
- **Matlab**, for the creation of the figures containing plots of functions throughout this document.

[1] `https://github.com/mamio1994/template-latex-unipd`
[2] `https://stackoverflow.com/questions/20530128/how-to-find-all-partitions-of-a-set`

Acknowledgements

As already mentioned, the idea behind this work comes from the group project developed for the *Networked Control for Multi-Agent System* course. During that group project, the problem of allocating quotas of a total amount through a distributed approach was posed, and the solution I later identified with the term *Exact-Breakdown method* was defined. I share the merit for that part with Marco Donà and Leonardo Gastaldello, my fellow students in that group, to whom I wish a very fulfilling conclusion of their studies and an excellent career afterwards. Also, that project would have been about a totally different matter without the contribution of Enel, my employer, who was eager to suggest a real business need to the group and to follow us along the project, especially in the person of Davide Passuello and the other colleagues who helped to retrieve useful information and data. Besides the people from Enel who cooperated in the mentioned project, I also want to thank all the colleagues from the team I currently belong to, who are always keen to establish a collaborative and friendly environment and to help each other out, providing some relief even when though challenges need to be addressed.

A special acknowledgement goes to Angelo Cenedese, the professor of the course that the group project was developed for and the supervisor of this thesis, for the inspiration he provided on the topics related to distributed control during the course and for his open-mindedness that allowed to explore a topic that is apparently not so covered from past studies.

A different project that contributed significantly to my education was the STAR Experiment, conducted in 2016 in the framework of the *Drop Your Thesis!* ESA programme, which was the first relevant international experience I had and the first time I was involved in the process of designing the electronics of a real device from scratch, making me more confident with the responsibilities that come with such a role. I owe an acknowledgement to Gilberto Grassi, the whole team of students and the academics who supported us, for giving me the chance to make

such a great experience, which has truly helped me in my professional growth.

I also want to mention the importance that open source communities had in the implementation part of this thesis, as nowawadays it is easier than ever to find online the solution to particular issues that one may encounter while developing. I strongly support open knowledge, as I think it is a practice that makes everybody win.

This thesis marks the end of my ten-year long journey as a working student. I would never have succeeded without the support of the wonderful and positive people I am surrounded with, who took care of helping me while I was standing stressful times, either by lending a hand on logistic tasks when I was busy or just by making me enjoy every moment of spare time I had, providing me with the recharge I needed to get through each stage of the route. My beloved friends are an invaluable source of energy, I strongly hope that we can continue to grow together for a long time to come.

A heartfelt acknowledgement goes also to my Mom, Dad and Brother, who made me the person I am and always offered their support beyond what I considered reasonable. I think that, without the inspiration they gave me during the first two decades of my life, I would not even have thought of trying to get a degree while working.

Last but definitely not least, I will never be grateful enough to Alice, with whom I shared every aspect of my life during last years. Her warm love and understanding were undoubtedly the keys for making it through the obstacles I faced in the final phases of my career as a student. She has constantly made me feel like everything is possible as long as we are together.

References

[1] R. Olfati-Saber, J. A. Fax, and R. M. Murray, "Consensus and cooperation in networked multi-agent systems," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 215–233, 2007.

[2] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279–301, 1989.

[3] M. H. DeGroot, "Reaching a consensus," *Journal of the American Statistical Association*, vol. 69, no. 345, pp. 118–121, 1974.

[4] M. Eremia and M. Shahidehpour, *Active Power and Frequency Control*, pp. 291–339. 2013.

[5] E. Fornasini, *Appunti di Teoria dei Sistemi*, ch. Complemento di Schur e inversione di una matrice a blocchi, pp. 594–595. Edizioni Libreria Progetto, 2015.

[6] *Building modern apps with Linux containers*. Red Hat, Inc.

[7] B. Payette and R. Siddaway, *Windows PowerShell in Action, Third Edition*. Manning Publications, 2017.

[8] R. C. Larson and A. R. Odoni, *Urban Operations Research*, ch. Generating samples from probability distributions. Prentice-Hall, 1981.