Synchronization and Fault-tolerance in Distributed Algorithms

Peva Blanchard

▶ **To cite this version:**

Peva Blanchard. Synchronization and Fault-tolerance in Distributed Algorithms. Distributed, Parallel, and Cluster Computing [cs.DC]. Université Paris Sud - Paris XI, 2014. English. NNT : 2014PA112219 . tel-01126873

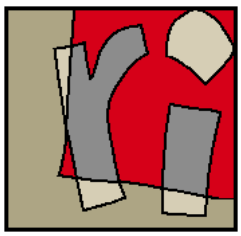
HAL Id: tel-01126873

https://theses.hal.science/tel-01126873

Submitted on 6 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Université Paris-Sud

École Doctorale 427 :
Informatique Paris-Sud

Laboratoire : LRI

Thèse

Informatique

par

Peva Blanchard

Synchronization and Fault-tolerance in Distributed Algorithms

Date de soutenance : 24/09/2014

Composition du jury :

Directeur de thèse :	M. Joffroy BEAUQUIER	Professeur (LRI, Paris XI)
Co-encadrante :	Mme. Sylvie DELAËT	Maître de Conférences (LRI, Paris XI)
Président du jury :	Christine PAULIN	Professeure (LRI, Paris XI)
Rapporteurs :	Rachid GUERRAOUI	Professeur (LPD, EPFL)
	Luis RODRIGUES	Professeur (DEI, Univ. de Lisboa)
Examinateurs :	Hugues FAUCONNIER	Maître de Conférences (LIAFA, Paris VII)
Invités :	Janna BURMAN	Maître de Conférences (LRI, Paris XI)

ἀλλ ὅτε δὴ πολύμητις ἀναιξειν Ὀδυσσεὺς
στάσκεν, ὑπαὶ δὲ ἴδεσκε κατὰ χθονὸς ὄμματα πήξας
σκῆπτρον δ΄ ὄυτ΄ ὀπίσω οὔτε προπρηνὲς ἐνώμα,
ἀλλ΄ ἀστεμπὲς ἔχεσκεν ἀίδρει φωτὶ ἐοικώς·
φαίης κε ζάκοτόν τέ τιν΄ ἔμμεναι ἄφρονά τ΄ αὔτως.
ἀλλ΄ ὅτε δὴ ὄπα τε μεγάλην ἐκ στήθεος εἵη
καὶ ἔππεα νιφάδεσσιν εοικότα χειμερίῃσιν
οὐκ ἂν ἔπειτ΄ Ὀδυσῆί γ΄ ἐρίσσειε βροτὸς ἄλλος·
οὐ τότε γ΄ ὥδ΄ Ὀδυσῆος ἀγασσάμεθ΄ εἶδος ἰδόντες.

Mais quand se leva Ulysse le subtil,
Il se tint d'abord immobile, les yeux fixés sur le sol,
Sans remuer son bâton, ni en avant, ni en arrière;
Il le gardait tout droit, comme un homme hébété;
On l'aurait cru quelqu'un qui s'est fâché, ou qui est sot.
Mais quand, de sa poitrine il laissa sortir sa grande voix,
Et des mots pareils à des flocons de neige en hiver,
Alors personne avec Uysse n'aurait pu rivaliser;
Et ce n'était plus l'allure d'Ulysse qui nous étonnait.

Iliade, III, v.216-224
trad. J.-L. Backès

Abstract

In the first part of this thesis, we focus on a recent model, called population protocols and introduced in [7], which describes large networks of tiny wireless mobile anonymous agents with very limited resources. The harsh constraints of the original model makes most of the classical problems of distributed algorithmics, such as data collection, consensus and leader election, either difficult to analyze or impossible to solve.

We first study the data collection problem, which mainly consists in transferring some values to a base station. By using a fairness assumption, known as cover times, introduced in [16], we compute tight bounds on the convergence time of concrete protocols. Next, we focus on the problems of consensus and leader election. It is shown that these problems are impossible in the original model. To circumvent these issues, we augment the original model with oracles, and study their relative power. We develop by the way a formal framework general enough to encompass various sorts of oracles, as well as their relations.

In the second part of the thesis, we study the problem of state-machine replication in the more classical model of asynchronous message-passing communication. The Paxos algorithm introduced in [56, 57] is a famous (partial) solution to the state-machine replication problem which tolerates crash failures. Our contribution is the enhancement of Paxos in order to tolerate transient faults as well. Doing so, we define the notion of practically self-stabilizing replicated state-machine.

Acknowledgements

First and foremost, I wish to thank Joffroy Beauquier for his dedication, guidance and invaluable insights, without which the present work would not have been possible.

I would also like to thank Sylvie Delaet for her generous support and help during this thesis.

I am gratefully thankful to Janna Burman for her strong dedication. Working with her has been a very rewarding experience.

I would like to thank Shlomi Dolev, from University Ben-Gurion, for our fruitful collaboration, as well as his generous hosting during my scientific visits in Beer-Sheeva.

Contents

Chapter 1

Introduction

The growing development of communication technologies, ranging from mobile sensor networks to georeplicated databases, have received considerable attention in the past few years. The goal of the distributed computing community is to provide the theoretical means to analyze and correctly use the variety of these communication networks. In particular, one of the fundamental aspects in distributed computing is the opposition *asynchrony vs. synchrony*, which will be studied in this thesis.

Let's first sketch why this opposition is ubiquitous in our domain. From a very general point of view, a distributed system, or network, is a collection of sequential processors together with communication abilities, e.g., message-passing, shared memory, etc. The usual sense of synchrony and asynchrony refers to the way processors' local transitions are interleaved. For example, in a synchronous system in the usual sense, each processor performs a local step at each global clock signal. On the other hand, in an asynchronous system, the processors perform their local steps almost independently of each other. This conrete meaning of synchrony and asynchrony is a particular case of a more general situation. *Asynchrony* is related to the relative independence of local transitions at processors, whereas *synchrony* is related to the relative dependence of distant processors among each other. We can associate with a given network, a specific *level of synchrony* that represent the extent to which the processors are relatively dependent on each other. Of course, this qualitative definition encompasses the usual quantitative definitions of synchrony (in terms of, e.g., delay, or periodicity), but it also applies to more general settings.

An important issue is that a problem in distributed computing usually consists in designing *local* algorithms for the processors to *coordinate* themselves, in order to perform a *global* task. Intuitively though, if the processors are too much independent from each other, then there is few chance that they can collaborate to perform the task. Put another way, if the level of synchrony provided by the network is too low, then the target problem may be impossible to solve. If the level of synchrony is fit, then a solution exists. And, if the level of synchrony is high, then a more efficient solution to the problem may exist.

There are essentially two ways to assess the level of synchrony of a network. First, we can adopt *explicit* assumptions: bounded communication delay, known movement pattern of mobile agents, known kinds of failures, etc. In that case, we usually adopt a bottom-up approach: we fix a level of synchrony

assuming explicit conditions on the network, and we design a distributed algorithm implementing the considered task. If the task turns out to be impossible for the given level of synchrony, one may also try to solve a weaker variant of the task.

Another approach consists in adopting *implicit* assumptions. In that case, the network is augmented with a black box, also known as an oracle, that provides some global service. This black box increases, in some sense, the level of synchrony of the network without referring to any inner mechanism. We then usually adopt a top-down approach: we fix the problem, and we look for the minimal level of synchrony required to solve[1] the problem.

Many fundamental results in our field of research illustrate the remarks above. The consensus problem fits perfectly in our discussion: intuitively, having all the processors to agree on a common value requires a relatively high level of synchrony. The seminal paper of Fischer, Lynch and Paterson [47] has shown that an asynchronous message-passing network prone to crash failures does not provide the required level of synchrony: a single crash failure may prevent the system to reach a consensus[2]. Later, Dwork, Lynch and Stockmeyer [45] adopted an explicit approach, and studied the consensus problem in partially synchronous message-passing networks prone to crash failures: in some cases, the consensus is solvable. In the same vein, some work focused on weaker variants of the consensus problem, e.g. [18]. An implicit approach to consensus was first presented in the seminal paper [32]. Instead of weakening the problem, or using explicit assumptions on the network, they introduced the concept of failure detector, i.e., an oracle that gives information about the past crash failures. By augmenting the network with such a device, they managed to solve the consensus problem. But their most important result is that they found the weakest failure detector, in their class of oracles, for solving the consensus problem [31]. Although the notion of weakest failure detector is problematic [33], this is the first occurrence of the implicit approach in distributed computing, as far as we know.

This thesis builds on both explicit and implicit approaches to solve classical distributed problems in two models. The first part concentrates on a model of large networks comprising tiny, resource-limited, anonymous and mobile agents, known as the population protocol model [7]. The second part, on the other hand, takes place in the asynchronous message-passing model, and studies the recently introduced relation [44] between self-stabilization and state-machine replication. In both cases, a close analysis of the tension between asynchrony and synchrony turns out to be the key for solving the considered problems.

[1] One may also look for the minimal level of synchrony to solve *efficiently* the problem, in some specific sense.

[2] A crash failure can occur at any time, independently of the other processors. The nature of the model implies that the crash is undetectable. Thus, the possibility of crash can be seen as an additional source of asynchrony in the general sense.

1.1 Population Protocols

The Original Model

The population protocols were introduced in [7] in order to model large networks of tiny, anonymous and asynchronous mobile agents. The basic idea is that the agents move passively, have a limited communication range, and two of them can communicate only if they are close enough (the two agents meet). The actual protocol consists in a finite set of rules describing how the states of two agents are updated when they meet. In this setting, a communication graph indicates the possibilities of meetings between the agents: a node represents an agent, and an edge the possibility of a meeting between its extremities. Time in population protocol is modeled by a scheduler, i.e., an adversary that selects the order in which the meetings occur. The mobility of the agents is modeled by a condition on this scheduler, usually referred to as a *fairness* condition. The original fairness condition, known as global fairness, somehow mimics a random scheduling of the meetings. Another important feature is that the agents have a memory size independent of the population size, and do not know which communication graph they are running on.

In terms of (a)synchrony, the original population protocol model is highly asynchronous. The scheduling of the meetings is non-deterministic, and some meeting between two agents can be arbitrarily delayed. Moreover, the restriction on the memory size implies that the agents do not have, and cannot compute, identifiers. Yet, the computational power of this model is not trivial. For instance, Angluin et al. [9] have proved that the class of computable predicates is exactly the Presburger class[3]. Later, the model began to be studied from a distributed computing perspective, involving topics like, e.g., fault tolerance, self-stabilization, leader election, mutual exclusion, and so on [10, 26, 36, 17]. Although some problems have a solution (e.g. self-stabilizing 2-hop coloring [10]), it turns out that many problems are impossible to solve in this model (e.g. silent leader election [27], self-stabilizing leader election [10]), and extensions of the original model have been proposed [15, 49, 64].

Most of the previous work rely on an explicit enhancement of the original model, and few adopted an implicit approach [65, 46]. A part of our contributions deal with the explicit approach: we apply an extension proposed in [16] to the study of the data collection problem. Our main contribution, however, is a formal framework which enables the correct manipulation of implicitly defined entities (oracles). We apply this framework to the study of consensus and (self-stabilizing) leader election in population protocols. The following sections give more details.

Explicit approach

As mentioned previously, the original model of population protocol is highly asynchronous. The original fairness condition allows a meeting between two agents to be delayed arbitrarily. This is problematic when one wants to analyze the speed of convergence of a protocol. In Chap. 4, we illustrate this issue by studying the data collection problem. In this problem, the agents are required

[3]However, their notion of computation does not require the termination of the computation. This highlights the highly asynchronous nature of the model.

to forward their initial values (e.g., given by sensor devices) to a base station. The convergence time is the time required to collect all the data at the base station. Many works [51, 54, 25] were dedicated to simulate data collection protocols in order to assess the convergence time. But simulations can only give hints on the performance of an algorithm, and the original population protocol model gives no analytical means to compute the convergence time.

Therefore, a new type of fairness, known as the *cover time* property, has been introduced in [16]. This property is an explicit condition on the scheduler that introduces a notion of partial synchrony (like in [40]). Roughly speaking, it guarantees that the agents meet periodically, some of them being faster than others.

This idea is applied to the study of a concrete example known as ZebraNet [53]. ZebraNet is a project conducted by the Princeton University and deployed in central Kenya. It aims at studying populations of zebras using sensors attached to the animals. This project developed an history-based protocol to deliver the sensed values to the base station. We incorporate "one-shot" variants (executing the data delivery only once) of this ZebraNet protocol in the population protocol model. Then, using the cover times, we give tight bounds on their convergence duration, expressed in number of agents meeting. As far as we know, this is the first analytical computation of convergence time of concrete population protocols.

Implicit approach

We illustrate the implicit approach by the study of two problems in distributed computing: consensus and leader election. In most cases, the highly asynchronous nature of the original population protocol model prevents the existence of solutions to both problems. This leads to the introduction of oracles, seen as the missing part of synchrony required to solve the considered problems. Investigating the nature of these oracles, and looking for the weakest ones, we were led to defining a whole framework to correctly manipulate these entities. For didactical reasons, we present these oracles in relation to the considered problems.

Consensus. In Chap. 5, we study the consensus problem: all the agents eventually decide on a common value among their initial values. The anonymous nature of the agents in the population protocol model naturally leads to define a variant of the consensus problem, namely the symmetric consensus, in which we additionally require that the decision value is stable under permutation of the initial distribution of the input values.

We first formally show that the consensus problem is impossible to solve, even without failures, in the original population protocol model. Our implicit approach consists in defining a class of oracles, similar to the failure detectors of Chandra and Toueg [32]. Roughly speaking, while a failure detector provides information about the failure pattern, our oracles provide information about the past schedule of meetings. In particular, we define an oracle, called $DejaVu$, which notifies some agent when it has indirectly seen every other, and we prove that it allows to solve the symmetric consensus. Next, similarly to [32], we say that an oracle O_1 is weaker than another oracle O_2 when there exists a population protocol that transforms the outputs of O_2 into possible

outputs of O_1. We then prove that the oracle $DejaVu$ is the weakest oracle (in its class) to solve the symmetric consensus.

Leader Election. The leader election problem is another classical problem in distributed algorithms, and consists in appointing a unique agent as the leader, while the others remain non-leaders. In the population protocol model, since there are no identifiers, the agents are not required to know *who* is the leader. To avoid trivial solutions, the agents start with the same initial state[4].

First, assuming that the agents are correctly initialized, we provide a protocol solving leader election over arbitrary communication graphs. As far as we know, no solutions were given over arbitrary graphs.

Next, we focus on the study of the self-stabilizing leader election problem. Self-stabilization, introduced by Dijkstra in [38], deals with transient faults, i.e., punctual corruptions of the states of the agents which put the whole system in an arbitrary configuration. The goal is to ensure that, after the last transient fault[5], the system eventually behaves correctly. This is equivalent to requiring that, in any fault-free execution from an *arbitrary* initial configuration, the system eventually behaves correctly.

It turns out that the possibility of transient faults weakens[6] the level of synchrony provided by the system to the point that the leader election problem becomes impossible in many natural cases [10]. To circumvent this issue, Fischer and Jiang [46] have introduced a new oracle, $\Omega?$, known as the *leader detector*, which basically notifies each agent about the presence of at least one leader in the system. Fischer and Jiang then have exhibited solutions using this oracle over the complete graphs, and over the rings.

Using the same oracle $\Omega?$, we build a self-stabilizing solution over the family of graphs with bounded degree. For a more general family of graphs, it seems that the oracle $\Omega?$ is not sufficient. We then introduce natural stronger variants of $\Omega?$, and use them to design a self-stabilizing protocol for leader election over arbitrary graphs.

As stated in the beginning, the implicit approach requires to look for the weakest oracle allowing the existence of a self-stabilizing leader election protocol. We prove that, over the rings, implementing $\Omega?$ is as hard as solving the self-stabilizing leader election problem. In particular, this implies that any oracle strong enough to yield a self-stabilizing leader election protocol, can be used to implement the oracle $\Omega?$.

Enhanced Model

In contrast to the oracles introduced for consensus, and to the failure detectors of Chandra and Toueg as well, the leader detector $\Omega?$ does not only observe the schedule of meetings, but also looks into the agent states and provides information about the absence or presence of leaders in the system. This introduces a kind of feedback loop, since the output of the oracle influences the agents behaviour, and vice-versa. This aspect deeply modifies the kind of manipulation allowed on oracles, and requires a proper framework.

[4] Except in the case of self-stabilization where the initial system's configuration is arbitrary.

[5] It is usually assumed that there is a finite number of transient faults.

[6] Roughly because the processors cannot detect the occurrence of a transient fault.

In Chap. 3, after presenting the population protocol model, we develop a natural framework that encompasses all the oracles mentioned above, as well as means to compare them. The basic idea is that a population protocol is a *local* piece of data, since it describes the states updates on meeting events. But, a problem, or an oracle, is a *global* piece of data, since it specifies the behaviour of the whole system. Naturally, a population protocol gives rise to a global behaviour, and this protocol is said to solve a problem if the associated behaviour matches with the problem specifications.

In our settings, oracles and problems live at the same level, and augmenting the network with an oracle is seen as allowing to compose the global behaviours of protocols with the oracle. Put another way, using an oracle to solve a problem is seen as a reduction from the latter to the former. This is analogous to the situation in complexity theory where one studies the complexity of a problem relatively to another one, by means of, e.g., deterministic polynomial time (sequential) algorithms.

In Chap. 3, we formally define the concepts of global behaviour, composition and comparison relation. As far as we know, this is the first framework unifying the various oracle-based approaches in population protocols.

1.2 State-Machine Replication and Self-Stabilization

The second part of this thesis takes place in the more classical asynchronous message-passing model with crash failures, and is dedicated to the study of state-machine replication. State-machine replication is a well-known technique to guarantee a fault-tolerant service [73]. The basic idea is to have many copies of the data, so that if some copies are lost, then the whole system is not broken. More precisely, each processor, also known as replica, holds a copy of the same program, or state-machine. The replicas start from the same initial state and have to execute the same requests in the same order. In other words, the replicas have to synchronize between themselves when the system receives requests from clients. Doing so, the clients see the whole system as a unique state-machine processing their requests in a sequential manner[7].

The Paxos algorithm, introduced by Lamport in [56, 57], is a *partial* solution to the state-machine replication problem: the algorithm only ensures that the replicas never "desynchronize", i.e., never exhibit incoherent answers to the clients; but, the replicas may, in some specific scenarios, undergo a livelock, preventing the system to execute new requests. This last issue is mainly due to the fact that the state-machine replication problem is related to the consensus problem[8], and it is well-known that the consensus problem is impossible in asynchronous message-passing networks with crash failures [47]. However, the livelock scenarios of Paxos can be avoided in practice using, e.g., failure detectors [32]. The usefulness of Paxos is proven daily by the very leading companies [30].

In our work, we enhance the original Paxos algorithm to make it *self-stabilizing*. In other words, our goal is to guarantee that, in any execution prone to crash failures and starting from an arbitrary configuration, the system eventually behaves correctly. Our approach replaces a core mechanism of

[7] The state-machine is then linearizable [50].
[8] The replicas have to agree on a common sequence of requests to execute.

Paxos, precisely its management of timestamps, with a more intricate structure, and we prove that the resulting algorithm eventually simulates the original Paxos for a practically infinite amount of time. In particular, our algorithm does not rely on additional assumptions to converge, and it requires the exact same level of synchrony from the system as the original Paxos to operate correctly after the convergence. As far as we know, this is the first attempt toward a self-stabilizing replicated state-machine in asynchronous message-passing.

1.3 Organization

The thesis is divided in two parts. The first part focuses on population protocols. In Chap. 3, we formally define the population protocols, along with several basic notions (schedule, fairness, etc.). We also introduce the notion of behaviour, which is used to model oracles and problems, as well as the notion of composition, and comparison of behaviours. In Chap. 4, we present our work on the data collection problem, giving a tight analysis of the convergence time of variants of the ZebraNet protocol. In Chap. 5, we turn to the implicit approach and study the consensus problem. In Chap. 5, we focus on the leader election problem, both with and without transient faults.

In the second part of the thesis, we concentrate on the problem of self-stabilizing replicated state-machine. Chap. 7 gives a general introduction to the problem. In Chap. 8, we give an informal description of our algorithm. The formal description is given in Chap. 9, and the detailed analysis and proofs are presented in Chap. 10.

Part I

Population Protocols

Chapter 2

Introduction

Nowadays, we see the rise of new kinds of networks comprising very tiny mobile sensors that can communicate with each other. The population protocol model [7] is a theoretical model that has been introduced to study the properties of these new networks. The basic idea of population protocols is to give rules telling what are the new states of meeting sensors based on their previous states. One may think of a population protocol as a set of chemical reactions that can occur when two chemical species come close enough.

This model also embeds two key aspects of mobile sensor networks. First, the sensors often move randomly in space, and two of them can communicate only when they are close enough, i.e., when they *meet*. In some cases, some sensors may never reach other sensors directly, or some sensors may move faster than others, the schedules of meetings may be governed by a probability distribution, etc. The variety of these cases leads to many sorts of mobility.

On the other hand, the sensors are usually small devices with very limited computational power. Also, it is possible that when the sensors are deployed, the exact number of active sensors and the communication possibilities are unknown. Therefore, the original population protocol model assumes the weakest hypothesis: the memory size of each sensor is independent of the size of the network. In particular, the sensors are anonymous; they cannot hold identifiers, nor compute them.

The original research on population protocols have focused on complexity issues: the goal was to determine what can be computed in this model [9]. For instance, imagine a population of penguins, each of them being equipped with a sensor able to tell whether its carrier is healthy or ill. At some point, every sensors make a measurement. One may then wonder, for instance, if it is possible for every sensors to collaborate and determine if a majority of penguins are healthy or not. This direction of research has received an interesting answer in [9]. Afterwards, the model was studied in relation with distributed algorithms issues, such as the classical problems of consensus, leader election, and so on. This line of research has required many modifications to the original model; some of which are original contributions of this thesis.

This part of the thesis proceeds as follows. We first give a detailed account of our model in Chap. 3. Then, in Chap. 4, we study the data collection problem. In Chap. 5, we examine the consensus problem, and, finally, in Chap. 6, the self-stabilizing leader election problem is analyzed.

Chapter 3

Model

3.1 Population Protocols

Population Protocols

The population protocols have been introduced to model networks of mobile and resource-limited agents. Roughly speaking, when two agents are close enough, they can communicate and update their states. The model do not describe explicitly the details of this communication. Indeed, the protocol is simply a set of rules telling what are the new states based on the previous states of the two meeting agents. The new states may additionally depend on inputs, e.g., from some sensor device, provided during the meeting. Moreover, the rules also specify the outputs produced by the agents during the meeting.

Formally, a *population protocol* $\mathcal{A}$ consists of a finite state space $States(\mathcal{A})$, a finite input alphabet $In(\mathcal{A})$, a finite output alphabet $Out(\mathcal{A})$, and a transition function $\delta : (States(\mathcal{A}) \times In(\mathcal{A}))^2 \to \mathcal{P}((States(\mathcal{A} \times Out(\mathcal{A}))^2)$ that maps any tuple (q_1, i_1, q_2, i_2) to a non-empty (finite) subset $\delta(q_1, i_1, q_2, i_2)$ in $(States(\mathcal{A}) \times Out(\mathcal{A}))^2$. The state space contains a set of *initial states* denoted by $InitStates(\mathcal{A})$. When it is clear from the context, we denote by $States$ (resp. In, Out) the state space (resp. input space, output space) of the protocol.

A *(transition) rule* of the protocol is a tuple $r = (q_1, i_1, q_2, i_2, q_1', o_1, q_2', o_2)$ such that $(q_1', o_1, q_2', o_2) \in \delta(q_1, i_1, q_2, i_2)$ and is denoted by $q_1, q_2 \xrightarrow[o_1,o_2]{i_1,i_2} q_1', q_2'$.

We refer to (q_1, i_1, q_2, i_2) (resp. (q_1', o_1, q_2', o_2)) as the *input side* (resp. *output side*) of the rule r.

The population protocol is *symmetric* when $q_1, q_2 \xrightarrow[o_1,o_2]{i_1,i_2} q_1', q_2'$ is a rule if and only if $q_2, q_1 \xrightarrow[o_2,o_1]{i_2,i_1} q_2', q_1'$ is a rule. The population protocol is *deterministic* if for every tuple (q_1, i_1, q_2, i_2), the set $\delta(q_1, i_1, q_2, i_2)$ has exactly one element. The population protocol is *output deterministic* if for every tuple (q_1, i_1, q_2, i_2), the set $\{(j_1, j_2),\ \exists(q_1', j_1, q_2', j_2) \in \delta(q_1, i_1, q_2, j_2)\}$ has exactly one element. All the population protocols we consider are output-deterministic. We will often assume, in the details of the proofs, that the outputs actually depend only on the states; this assumption is acceptable since it is possible to encode the output in the states.

Communication Graphs

The set of rules as above only specify the protocol. To model the mobility of a set of agents, a *communication graph* is required. The nodes of this graph represent the mobile agents. Contrary to classical communication graphs, an edge between two agents does not represent a physical communication link, but only the *possibility of a meeting* between the two agents. For instance, one may imagine a population of animals, such that each of them only visit specific places. Then according their respective set of visited places, two animals in the population may never meet.

Formally, a *communication graph* is represented by a weakly connected directed graph G. We denote by $Vert(G)$ (resp. $Edges(G)$) the set of vertices (resp. of edges). Each vertex represents a finite-state sensing device called an *agent*, and an edge (x, y) indicates the possibility of a communication between x and y in which x is the *initiator* and y is the *responder*. The orientation of an edge corresponds to this asymmetry in the communications.

Schedules

A communication graph describe the possibility of interactions between the agents. In the population protocol model, a *meeting event* represents the meeting of two agents. Note that the model precludes the simultaneous meeting of more than two agents. A *schedule* then simply consists a sequence of events.

Formally, given a communication graph G, a *meeting event* is represented by an edge of G. We denote by $x \in e$ the fact that the vertex x is *involved* in e, i.e., e is incident to x. We denote by $e \cap e'$ the set of agents that are involved in both events e and e'. We define an *independence relation* on $Edges(G)$ as follows: two events e and e' are independent if and only if they involve no common agent, i.e., $e \cap e' = \emptyset$.

A *schedule* S is a sequence $S = (e_t)_{0 \leq t < T}$ ($T \in \mathbb{N} \cup \{\infty\}$) of events. An *event occurrence in* S is a couple $(t, e) \in \mathbb{N} \times \Sigma$ such that $e_t = e$. We often refer to an event occurrence in a schedule simply as an event in this schedule. The *support* of a schedule S is the set of agents, denoted by $supp(S)$, that are involved in the events occurring in S.

If S is a finite schedule, and S' a finite or infinite schedule, we denote by $S \cdot S'$, or simply SS', the concatenation of these schedules. A *prefix* (resp. a *factor*) of the schedule S is a schedule K such that $S = K \cdot B$ for some schedule B (resp. $S = A \cdot K \cdot B$ for some schedules A,B). If p is an event occurrence in S, we denote by $S \uparrow p$ the prefix of S that ends with the event occurrence p.

Let $S = (e_t)_{0 \leq t < T}$ be a (possibly infinite) schedule, and $\alpha \in \mathfrak{S}G$ be an automorphism of the underlying graph G, and $S = e_0 e_1 \dots$ be any schedule on this graph. We denote by αS the schedule $(\alpha(e_t))_{0 \leq t < T}$. Let $\tau \in \mathfrak{S}[0, T)$ be a permutation of the indices. We denote by $S\tau$ the schedule $(e_{\tau(t)})_{0 \leq t < T}$.

Each schedule S yields a partially ordered set $\mathbb{P}(S)$ as follows. The elements of $\mathbb{P}(S)$ are the event occurrences of S. The partial order is the transitive closure of:

$$(t, e) \rightsquigarrow (t', e') \Leftrightarrow t \leq t' \wedge e \cap e' \neq \emptyset \tag{3.1}$$

We say that (t', e') *causally depends on* (t, e). We refer to this order as the *causal order on* S. This order simply comes from the fact that, if two events involve the same agent, then one of the events must causally precede the other.

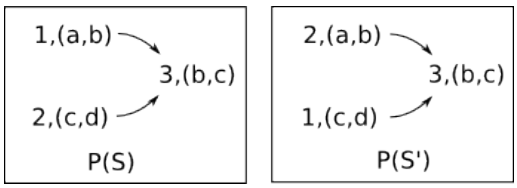


Figure 3.1: Causal diagrams – $S = (a,b)(c,d)(b,c)$, $S' = (c,d)(a,b)(b,c)$.

The poset $\mathbb{P}(S)$ can be seen as a *causal diagram* describing the causal relations between the event occurrences in S (Fig. 3.1).

Actually, we see in Fig. 3.1, that this partial order allows to define an equivalence relation on schedules. Intuitively, two schedules S and S' are causally equivalent, denoted by $S \simeq S'$, if their causal diagrams look the same. Put another way, S' is obtained from S by permuting its event occurrences in a way that respects causality.

Formally, two schedules $S = (e_t)_{0 \leq t < T}$ and $S' = (e'_t)_{0 \leq t < T}$ are *causally equivalent* if there exists a permutation $\tau \in \mathfrak{S}[0,T)$ of the indices such that $S' = S\tau$, and, for every t, t', $(t, e_{\tau(t)}) \rightsquigarrow (t', e_{\tau(t')})$ in $\mathbb{P}(S\tau)$ if and only if $(\tau(t), e_{\tau(t)}) \rightsquigarrow (\tau(t'), e_{\tau(t')})$ in $\mathbb{P}(S)$.

Thanks to this equivalence relation, we can define weaker notions of prefixes, and factors. A schedule K is a *commuting prefix* (resp. *commuting factor*) of a schedule S if $S \simeq K \cdot B$ (resp. $S \simeq A \cdot K \cdot B$) for some schedule B (resp. schedules A and B). In other words, a commuting prefix (resp. commuting factor) is a prefix (resp. factor) up to equivalence.

Given an event occurrence p in S, we define the *causal past* (resp. *causal future*) of p as the sub-poset of $\mathbb{P}(S)$ comprising the event occurrences on which p causally depends (resp. comprising the event occurrences that causally depend on p). We denote by $Past(p, S)$ (resp. $Future(p, S)$), or simply $Past(p)$ (resp. $Future(p)$), the causal past (resp. causal future) of p in S.

$$Past(p) = \{e \text{ in } S,\ e \rightsquigarrow p\} \tag{3.2}$$
$$Future(p) = \{e \text{ in } S,\ p \rightsquigarrow e\} \tag{3.3}$$

A finite schedule K is a *past cone* (resp. *future cone*) if there exists an event occurrence p in K such that all the event occurrences in K are in the causal past (resp. causal future) of p. If p involves an agent x, K is said to be a past cone (resp. future cone) *at* x.

Histories

Given a schedule of events, an external observer (out of the system) may observe the inputs provided to (or the outputs produced by) two agents during a meeting. For instance, if one is interested in electing a leader, one will only focus on the leader bit output by the agents, and will not burden herself with all the details of the agents states. The concept of *history* is introduced to model sequences of values attached to the events of a schedule. One may also think of a history a schedule *augmented with* values from some domain.

Formally, a *history with values in set R* is a couple $H = (S, h)$ where S is a schedule, and h is function that associates with each occurrence of meeting event $e = (x, y)$, a couple (i_x, i_y) of values in R. The value i_x (resp. i_y) is the *output of the history at x (resp. at y) in event e*. The schedule S is the *underlying schedule* of the history H.

Given a prefix, or a factor, K of the schedule S, the *restriction of H to K* is the history, denoted by $H|_K$, with schedule K which outputs the same values as H during K.

Given $\alpha \in \mathfrak{S}G$ an automorphism of the communication graph, we denote by αH the history $(\alpha S, h')$ where the function h' maps $(t, \alpha(e_t))$ to the value $h(t, e_t)$. In other words, αH is the history obtained from H by renaming the agents according to α.

Given $\alpha \in \mathfrak{S}[0, T)$ ($T \in \mathbb{N} \cup \{\infty\}$) a permutation of the natural numbers, we denote by $H\tau$ the history $(S\tau, h')$ where h' maps $(t, e_{\tau(t)})$ to the value $h(\tau(t), e_{\tau(t)})$. In other words, $H\tau$ is obtained from H by permuting the order of events in S along with their associated history values.

Assignments, Traces

With the definition of history, the values at some agent are provided during a meeting event involving this agent. It is sometimes more convenient to record at each agent the last value output by the history.

First, an *assignment with values in a set R* is a function that assigns to each agent of the communication graph a value from R.

Then, we define the *trace T associated with a history $H = (S, h)$* defined as follows. For every non-empty finite prefix K of S, for every agent x, $T(K, x)$ is the last value output at x by H during K if this value exists, or is left undefined otherwise. We denote by $T(K)$ the assignment that assigns the value $T(K, x)$ to each agent x. Note that if $e = (x, y)$ is a meeting event, then $T(K \cdot e)$ and $T(K)$ may only differ at x and y. In other words, a trace is a sequence of assignments such that two consecutive assignments may only differ at the agents involved in the corresponding meeting event.

A trace is *constant* if all the assignments are equal. A trace is *uniform* if all the assignments are equal and assign the same value to every agent in the system. It is not difficult to see that a history is entirely determined by its trace, and thus, the two formulations are equivalent. Similarly, a history is constant (resp. uniform) if its associated trace is constant (resp. uniform).

Executions

Basically, given a configuration, one can first select two agents (an edge of the communication graph), and input values for them, and then apply the corresponding rule of the protocol (or one matching rule if the protocol is non-deterministic) to get a new configuration. An execution is simply the repetition of such choices. Equivalently, an execution can be defined as a sequence of configurations along with a history with values in the input alphabet, and a history with values in the output alphabet (and the same underlying schedule); the configurations between related to each other by transition with inputs and outputs given by the histories.

Formally, consider a population protocol $\mathcal{A}$. A *configuration* is an assignment $\gamma : Vert(G) \to States$ specifying the state of each agent in the network.

An *action* is a couple $\sigma = (e, r)$ where $e = (x, y)$ is an event, and $r : q_1, q_2 \xrightarrow[o_1,o_2]{i_1,i_2} q_1', q_2'$ is a rule of the protocol. The couple (i_1, i_2) (resp. (o_1, o_2)) is the *input (resp. output) values associated with the action* σ.

Given two configurations γ, γ', we say that *the configuration* γ *goes to* γ' *via the action* σ when $(\gamma(x), \gamma(y)) = (q_1, q_2)$, $(\gamma'(x), \gamma'(y)) = (q_1', q_2')$ and for all $z \notin \{x, y\}$, $\gamma(z) = \gamma'(z)$; we denote such a relation by $\gamma \xrightarrow{\sigma} \gamma'$.

An *execution* of $\mathcal{A}$ is a tuple $((\gamma_t)_{t\in\mathbb{N}}, H_{in}, H_{out})$ where the γ_t are configurations, $H_{in} = (S, h_{in})$ and $H_{out} = (S, h_{out})$ are histories with the same underlying schedule $S = (e_t)_{0\leq t<T}$ and with values in $In(\mathcal{A})$ and $Out(\mathcal{A})$ respectively, such that, for all t, there exists an action σ_t with event e_t, input values $h_{in}(t, e_t)$ and output values $h_{out}(t, e_t)$ such that $\gamma_t \xrightarrow{\sigma_t} \gamma_{t+1}$. It is also assumed that the first configuration γ_0 contains initial states only.

We say that an action $\sigma = (e, q_1, q_2 \xrightarrow[o_1,o_2]{i_1,i_2} q_1', q_2')$ is *enabled at time* t *in* E if $e_t = e$, $(\gamma_t(x), \gamma_t(y)) = (q_1, q_2)$, and $h_{in}(t, e_t) = (i_1, i_2)$.

We say that the action $\sigma = (e, q_1, q_2 \xrightarrow[o_1,o_2]{i_1,i_2} q_1', q_2')$ is *triggered at time* t *in* E if $e_t = e$, $(\gamma_t(x), \gamma_t(y)) = (q_1, q_2)$, $(\gamma_{t+1}(x), \gamma_{t+1}(y)) = (q_1', q_2')$, $h_{in}(t, e_t) = (i_1, i_2)$ and $h_{out}(t, e_t) = (o_1, o_2)$.

The history H_{in} (resp. H_{out}) is the *input history* (resp. *output history*) of the execution. A priori, the protocol is non-deterministic, hence, there may be many executions with the same input history H_{in} and the same starting configuration γ_0. We denote by $H_{in}[\gamma_0]$ the set of these executions. When the protocol is deterministic, $H_{in}[\gamma_0]$ contains a unique execution which is also denoted by $H_{in}[\gamma_0]$.

In some cases, it is simpler to deal with traces instead of histories. In that case, an execution E is equivalently pictured as a sequence $(\gamma_t, \alpha_t, \beta_t)_{0\leq t<T}$ where the γ_t's are the configurations, $T_{in} = (\alpha_t)_{0\leq t<T}$ is the trace corresponding to the input history (the input trace), and $T_{out} = (\beta_t)_{0\leq t<T}$ is the trace corresponding to the output history (the output trace). Moreover, the output is often encoded in the states of the agents. In that case, the output trace T_{out} is completely determined by the sequence of configurations $(\gamma_t)_{O\leq t<T}$, and an execution is simply pictured as the sequence $(\gamma_t, \alpha_t)_{0\leq t<T}$.

Composition of Population Protocols

It is often desirable to split an object in simpler elements. Once each element is defined, one can combine them to build the target object. The same is true for population protocols. In this section, we define three basic operations that syntactically produce new protocols from given ones.

For sake of readability, an element (x, y) in the cartesian product $X \times Y$ is often denoted by $x \cdot y$. Let $\mathcal{A}$ and $\mathcal{B}$ be two population protocols.

(Parallel). Intuitively, the parallel composition corresponds to running simultaneously two protocols. Hence, the new state space is the cartesian product of the two previous state spaces, and the same for the input/output alphabets mutatis mutandis. A transition rule in the new protocol corresponds to

the conjunction of a rule from the first protocol and a rule from the second protocol.

More formally, the *parallel composition* of $\mathcal{A}$ and $\mathcal{B}$ is the population protocol $\mathcal{C}$ defined as follows.

$$States(\mathcal{C}) = States(\mathcal{A}) \times States(\mathcal{B}) \tag{3.4}$$
$$In(\mathcal{C}) = In(\mathcal{A}) \times In(\mathcal{B}) \tag{3.5}$$
$$Out(\mathcal{C}) = Out(\mathcal{A}) \times Out(\mathcal{B}) \tag{3.6}$$

For $k = 1, 2$, let $a_k, a'_k \in States(\mathcal{A})$, $b_k, b'_k \in States(\mathcal{B})$, $ia_k \in In(\mathcal{A})$, $oa_k \in Out(\mathcal{A})$, $ib_k \in In(\mathcal{B})$ and $ob_k \in Out(\mathcal{B})$. The rule

$$a_1 \cdot b_1, a_2 \cdot b_2 \xrightarrow[oa_1 \cdot ob_1, oa_2 \cdot ob_2]{ia_1 \cdot ib_1, ia_2 \cdot ib_2} a'_1 \cdot b'_1, a'_2 \cdot b'_2 \tag{3.7}$$

is a rule of $\mathcal{C}$ if and only if

$$a_1, a_2 \xrightarrow[oa_1, oa_2]{ia_1, ia_2} a'_1, a'_2 \text{ in } \mathcal{A} \tag{3.8}$$
$$b_1, b_2 \xrightarrow[ob_1, ob_2]{ib_1, ib_2} b'_1, b'_2 \text{ in } \mathcal{B} \tag{3.9}$$

This new protocol is denoted by $\mathcal{C} = \mathcal{A} \otimes \mathcal{B}$.

(Serial). Intuitively, the serial composition corresponds to plugging the values output by a protocol as input values to another protocol. Obviously, this is possible if and only if the former's output and the latter's input alphabets are the same. The transmission of the values is supposed to be instantaneous.

More formally, when $Out(\mathcal{B}) = In(\mathcal{A})$, the *serial composition* of $\mathcal{A}$ and $\mathcal{B}$ is the protocol $\mathcal{C}$ defined as follows.

$$States(\mathcal{C}) = States(\mathcal{A}) \times States(\mathcal{B}) \tag{3.10}$$
$$In(\mathcal{C}) = In(\mathcal{B}) \tag{3.11}$$
$$Out(\mathcal{C}) = Out(\mathcal{A}) \tag{3.12}$$

For $k = 1, 2$, let $a_k, a'_k \in States(\mathcal{A})$, $b_k, b'_k \in States(\mathcal{B})$, $i_k \in In(\mathcal{B})$, $o_k \in Out(\mathcal{A})$. The rule

$$a_1 \cdot b_1, a_2 \cdot b_2 \xrightarrow[o_1, o_2]{i_1, i_2} a'_1 \cdot b'_1, a'_2 \cdot b'_2$$

is a rule of $\mathcal{C}$ if and only if there exists $j_1, j_2 \in Out(\mathcal{B}) = In(\mathcal{A})$ such that

$$b_1, b_2 \xrightarrow[j_1, j_2]{i_1, i_2} b'_1, b'_2 \text{ in } \mathcal{B} \tag{3.13}$$
$$a_1, a_2 \xrightarrow[o_1, o_2]{j_1, j_2} a'_1, a'_2 \text{ in } \mathcal{A} \tag{3.14}$$

This new protocol is denoted by $\mathcal{C} = \mathcal{A} \circ \mathcal{B}$.

(Feedback). The feedback composition operates on a single protocol, and corresponds to plugging a part of its output to its own input.

Formally, assume that $In(\mathcal{A}) = U \times I$ and $Out(\mathcal{A}) = U \times O$. The *feedback composition of $\mathcal{A}$ along U* is the protocol $\mathcal{C}$ defined as follows.

$$States(\mathcal{C}) = States(\mathcal{A}) \tag{3.15}$$

$$In(\mathcal{C}) = I \tag{3.16}$$

$$Out(\mathcal{C}) = O \tag{3.17}$$

For $k = 1, 2$, let $a_k, a'_k \in States(\mathcal{A})$, $b_k, b'_k \in States(\mathcal{B})$, $i_k \in I$, $o_k \in O$. The rule

$$a_1, a_2 \xrightarrow[o_1,o_2]{i_1,i_2} a'_1, a'_2 \tag{3.18}$$

is a rule of $\mathcal{C}$ if and only if there exist $u_1, u_2 \in U$ such that

$$a_1, a_2 \xrightarrow[u_1 \cdot o_1, u_2 \cdot o_2]{u_1 \cdot i_1, u_2 \cdot i_2} a'_1, a'_2 \text{ in } \mathcal{A} \tag{3.19}$$

This new protocol is denoted by $\mathcal{C} = Feedback_U(\mathcal{A})$, or simply $\mathcal{C} = Feedback(\mathcal{A})$ when U is clear from the context.

3.2 Fairness

Fix a communication graph G. In the definition of the executions above, no constraints have been given. As a consequence, for instance, the execution obtained by selecting the same pair of agents forever is possible. Therefore, in order to preclude such pathological cases, one must define a criterion, usually known as a *fairness condition*. In this section, we present several such criteria, and thus, several kinds of *fair executions*. Note that, some of these notions yield constraints on the underlying schedules only, while others formulas also involve the sequence of configurations in the execution.

Classical Fairness

A schedule S is *classically fair* when, for every edge (x, y) of the communication graph, the event (x, y) (x initiator, y responder) occurs infinitely often in S.

An execution is classically fair when its underlying schedule is. Thus, this fairness condition only relies on the underlying schedule of the execution.

This fairness condition is not used very often, but it represents the most natural definition of fairness. It is introduced here for didactical reasons as a point of reference for the other fairness conditions.

Weak Fairness

The following fairness condition is a slightly weaker form of the classical fairness in the sense that the agents are not required to meet *directly* infinitely often. They are only required to meet *indirectly*, i.e., via the mediation of other agents.

Formally, consider a schedule S, a segment K of S, and two agents x and y. We say that *y meets indirectly with x during K* if there are occurrences of meeting events $e_1, \ldots, e_s$ during u such that e_i occurs before e_{i+1} and:

$$x \in e_1 \tag{3.20}$$

$$e_i \cap e_{i+1} \neq \emptyset,\ 1 \leq i < s \tag{3.21}$$

$$y \in e_s \tag{3.22}$$

In other words, e_1 involves x, e_s involves y, and we have the relations $e_1 \rightsquigarrow \cdots \rightsquigarrow e_s$ in $\mathbb{P}(K)$. Note that this notion is not symmetric.

A schedule S is *weakly fair* when every agent meets indirectly with every other agents infinitely often during S. In other words, for each (ordered) pair of agents (x, y), S contains infinitely many segments K during which y meets indirectly with x.

An execution is weakly fair when its underlying schedule is. Thus, this fairness condition only relies on the underlying schedule of the execution.

Cover Times

The following fairness condition introduce an idea similar to the *partial synchrony* from [45]. This notion stems from the fact that in many real mobile networks, the agents meet according to an almost regular pattern. In many works [25, 29, 1, 51, 54], it is observed that the time between two meetings of two agents x and y is bounded. The *cover time property* inspires from this observation by attaching to every agent x an integer, known as the *cover time of the agent* x, that represents the time (counted in number of meeting events) required for x to meet every other agent at least once. This fairness condition was first presented in [16].

More formally, we assume that the vertices of the communication graph are labeled by positive integers. The integer cv_x associated with the agent x is called the *cover time of agent* x. The vector $(cv_x)_{x \in V}$ is called the *cover time vector*.

A schedule S is said to satisfy the *cover time property* if, for every agent x, for every segment K in S of size greater than or equal to cv_x, x meets at least once with every other agents during K.

Again, an execution satisfies the cover time property if its underlying schedule does.

Global Fairness

Contrary to the previous cases, the following fairness condition [7] is stated with a reference to the protocol; it does not rely only on the underlying schedule of the execution. The idea is to mimic the properties of a random walk: if a configuration is infinitely often reachable, then this configuration is reached infinitely often.

More formally, an execution $E = ((\gamma_t)_{t \in \mathbb{N}}, H_{in}, H_{out})$ is *globally fair* if for every configuration γ and couple (i_1, i_2) of input values such that $(\gamma_t, H_{in}(t)) = (\gamma, (i_1, i_2))$ infinitely often, if there exists a configuration γ' and an action σ with input values (i_1, i_2) such that $\gamma \xrightarrow{\sigma} \gamma'$, then γ' occurs infinitely often in E.

Local Fairness

The global fairness condition is related to the reachability of configurations, but it does not tell anything about how the configurations are reached, i.e., about the actual transitions that lead to these configurations. For instance, one can imagine a situation where there are at least two actions σ_1 and σ_2 that both lead to a configuration γ' from a configuration γ. Roughly speaking, if

γ occurs infinitely often in the execution, the global fairness only ensures that γ' is reached infinitely often, but it may happen that the transition $\gamma \xrightarrow{\sigma_2} \gamma'$ is never triggered.

Therefore, the following "orthogonal" definition was introduced [46]: an execution E is *locally fair* if for every possible action σ, if σ is enabled infinitely often during E, then σ is triggered infinitely often during E.

As for the global fairness, the local fairness refers to the protocol; it does not rely only on the underlying schedule.

3.3 Behaviours

In the previous sections, the population protocols have been defined. Roughly speaking, a population protocol is a *local* data because it describes the behaviour of the agents at the level of pairwise interaction. However, most of the problems such as, e.g., leader election, are stated from a global point of view: one is only interested in the possible output histories of the execution given some input history (if any). In addition, one may want to solve some problem by assuming that some oracle is available.

In this section, the notion of *behaviour* is introduced to model this global point of view. This notion is general enough to model both problems, and oracles. In the sequel, it is shown how one can associate a behaviour (global data) with a population protocol (local data).

Definitions

Intuitively, a *behaviour* is simply a relation between input histories and output histories. One can think of it as a specification relating the legal output histories given an input history.

Precisely, a behaviour B is given by a family $Dom(B)$ of graphs (the domain of B), an input alphabet $In(B)$, an output alphabet $Out(B)$ and a function that maps any graph G in $Dom(B)$, and any history $H_{in} = (S, h_{in})$ with values in $In(B)$ to a set $B(G, H_{in})$ of histories $H_{out} = (S, h_{out})$ with values in $Out(B)$ and the same underlying schedule S.

Composition of Behaviours

Like population protocols, the behaviours have inputs and outputs. Therefore, we can define the same three operations (parallel, serial, feedback) of composition on the behaviours. These operations have the same meaning as in the case of population protocols.

Let A and B be two behaviours with the same domain of graphs, i.e., $Dom(A) = Dom(B)$.

(Parallel). First, the *parallel composition* of A and B is the behaviour C defined as follows.

$$Dom(C) = Dom(A) = Dom(B) \tag{3.23}$$

$$In(C) = In(A) \times In(B) \tag{3.24}$$

$$Out(C) = Out(A) \times Out(B) \tag{3.25}$$

The history $(S, (h^A_{out}, h^B_{out}))$ belongs to $C(G, (S, (h^A_{in}, h^B_{in})))$ if and only if

$$(S, h^A_{out}) \in A(G, (S, h^A_{in})) \tag{3.26}$$
$$(S, h^B_{out}) \in B(G, (S, h^B_{in})) \tag{3.27}$$

We denote this behaviour by $C = A \otimes B$.

(Serial). When $Out(B) = In(A)$, the *serial composition* of A and B is the behaviour C defined as follows.

$$Dom(C) = Dom(A) = Dom(B) \tag{3.28}$$
$$In(C) = In(B) \tag{3.29}$$
$$Out(C) = Out(A) \tag{3.30}$$

The history (S, h^A_{out}) belongs to $C(G, (S, h^B_{in}))$ if and only if there exists a history (S, h) such that

$$(S, h^A_{out}) \in A(G, (S, h)) \tag{3.31}$$
$$(S, h) \in B(G, (S, h^B_{in})) \tag{3.32}$$

We denote this behaviour by $C = A \circ B$.

(Feedback). When $In(A) = U \times I$ and $Out(A) = U \times O$ for some sets U, I, O, the *feedback composition of A along U* is the behaviour C defined as follows.

$$Dom(C) = Dom(A) \tag{3.33}$$
$$In(C) = I \tag{3.34}$$
$$Out(C) = O \tag{3.35}$$

The history (S, h_{out}) belongs to $C(G, (S, h_{in}))$ if and only if there exists a history (S, h) with values in U such that

$$(S, (h, h_{out})) \in A(G, (S, (h, h_{in}))) \tag{3.36}$$

This new behaviour is denoted by $C = Feedback_U(A)$, or simply $C = Feedback(A)$ when U is clear from the context.

Sub-behaviour

Consider two behaviours A and B with the same domain $Dom(A) = Dom(B)$, same input alphabet $In(A) = In(B)$ and same output alphabet $Out(A) = Out(B)$. We say that B *is a sub-behaviour of* A, denoted by $B \subseteq A$ or $A \supseteq B$, when $B(G, H) \subseteq A(G, H)$ for every graph G in the domain, and every history H on G with values in the input alphabet.

Intuitively, the behaviour B is "stronger" than the behaviour A, because, given the same input history, the legal output histories of B are also legal output histories of A. Another way to see this is to consider the trivial behaviour that associates with each input history the set of all possible histories with values in the output alphabet. It becomes clear that any behaviour (with same alphabets) is a sub-behaviour of this trivial behaviour.

3.4 Behaviour associated with a Population Protocol

Intuitively, the behaviour of a population protocol sums up the observable part of its executions; i.e., we forget about the states of the agents, and only focus on the input and output histories yielded by the executions. However, we do not always consider every possible execution: we generally fix a class of execution, or *context*, and observe the input and output histories for such a class.

In the sequel, we show that, if the considered context is *acceptable* (see Def. 1), then the behaviour associated with the composition of protocols implements the composition of the behaviours associated with the population protocols.

Context

A *context* $\mathbb{C}$ is a map which associates with every graph G from some family of graphs and every population protocol $\mathcal{A}$, a set of executions of $\mathcal{A}$ on G. The context $\mathbb{C}$ is usually defined by

- a family $Dom(\mathbb{C})$ of graphs.
- an initialization map telling what the possible initial configurations are on any given graph from $Dom(\mathbb{C})$
- a fairness condition telling which executions are assumed to be fair.

The executions defined by $\mathbb{C}$ on a given graph G are referred to as the $\mathbb{C}$-*legal executions on* G, or simply the *legal executions on* G.

Behaviour associated with a protocol

The *behaviour of the protocol* $\mathcal{A}$ *under the context* $\mathbb{C}$ is the behaviour B defined as follows. First

$$Dom(B) = Dom(\mathbb{C}) \tag{3.37}$$
$$In(B) = In(\mathcal{A}) \tag{3.38}$$
$$Out(B) = Out(\mathcal{B}) \tag{3.39}$$

We have $(S, h_{out}) \in B(G, (S, h_{in}))$ if and only if there exists a $\mathbb{C}$-legal execution of $\mathcal{A}$ on G with schedule S, input history (S, h_{in}) and output history (S, h_{out}). This behaviour is denoted by $B = Beh_{\mathbb{C}}(\mathcal{A})$, or, when the context is obvious, simply $B = Beh(\mathcal{A})$.

Structure Theorems

Prop. 1 shows that the different composition operations (parallel, serial, feedback) for behaviours are compatible with the sub-behaviour relation. Prop. 2 shows that, given an acceptable context (see Def. 1), the map that associates with each protocol $\mathcal{A}$ the behaviour $Beh(\mathcal{A})$ almost preserves the composition operations.

Proposition 1. *Let A and B be sub-behaviours of C and D respectively. Then, whenever the composition is defined,*

- *$A \otimes B$ is a sub-behaviour of $C \otimes D$.*
- *$A \circ B$ is a sub-behaviour of $C \circ D$.*
- *$Feedback(A)$ is a sub-behaviour of $Feedback(C)$.*

Proof. These claims are straightforward from the definitions. □

Definition 1 (Acceptable Context)**.** *A context $\mathbb{C}$ is* acceptable *if for any graph G, any protocols $\mathcal{A}$ and $\mathcal{B}$, the following holds:*

- *the set of initial configurations of $\mathcal{A} \otimes \mathcal{B}$ (resp. $\mathcal{A} \circ \mathcal{B}$) is exactly the cartesian product of the sets of initial configurations of $\mathcal{A}$ and $\mathcal{B}$.*
- *the set of initial configurations of $Feedback(\mathcal{A})$ is exactly the set of initial configurations of $\mathcal{A}$.*
- *A fair execution of $\mathcal{A} \otimes \mathcal{B}$ (or $\mathcal{A} \circ \mathcal{B}$) yields fair executions of $\mathcal{A}$ and $\mathcal{B}$.*
- *A fair execution of $Feedback(\mathcal{A})$ yields fair executions of $\mathcal{A}$.*

The first two points basically states that initializing the composed protocol is equivalent to initializing each component separately. The last points state that, in a fair execution of the composed protocol, everything looks like a fair execution from the point of view of each component.

Proposition 2. *Fix an acceptable context $\mathbb{C}$. Let $\mathcal{A}, \mathcal{B}$ be population protocols. Then, whenever it is defined:*

- *$Beh(\mathcal{A} \otimes \mathcal{B})$ is a sub-behaviour of $Beh(\mathcal{A}) \otimes Beh(\mathcal{B})$*
- *$Beh(\mathcal{A} \circ \mathcal{B})$ is a sub-behaviour of $Beh(\mathcal{A}) \circ Beh(\mathcal{B})$*
- *$Beh(Feedback(\mathcal{A}))$ is a sub-behaviour of $Feedback(Beh(\mathcal{A}))$.*

Proof. We study the cases separately. In each of them, C denotes the first behaviour, and D the second one, such that the claim is C being a sub-behaviour of D.

(parallel). Let $H_{out} = (S, (a_{out}, b_{out}))$ be an output history of C corresponding to the input history $H_{in} = (S, (a_{in}, b_{in}))$. We have to prove that $H^a_{out} = (S, a_{out})$ (resp. $H^a_{out} = (S, b_{out})$) is an output history of $Beh(\mathcal{A})$ (resp. $Beh(\mathcal{B})$) corresponding to the input history $H^a_{in} = (S, a_{in})$ (resp. $H^b_{in} = (S, b_{in})$).

By definition, there exists a $\mathbb{C}$-legal execution E of $\mathcal{A} \otimes \mathcal{B}$ with the schedule S, input history H_{in} and output history H_{out}. The execution E can be written

$$\dots (\gamma^a_t, \gamma^b_t) \xrightarrow{\sigma_t} (\gamma^a_{t+1}, \gamma^b_{t+1}) \dots \tag{3.40}$$

where γ^a_* (resp. γ^b_*) are configurations of $\mathcal{A}$ (resp. $\mathcal{B}$), and $\sigma_t = (e_t = (x_t, y_t), r_t)$ is an action such that the rule r_t of $\mathcal{A} \otimes \mathcal{B}$ is equivalent to

$$r^a_t : \ \gamma^a_t(x_t), \gamma^a_t(y_t) \xrightarrow[a_{out}(e_t)]{a_{in}(e_t)} \gamma^a_{t+1}(x_t), \gamma^a_{t+1}(y_t) \tag{3.41}$$

$$r^b_t : \ \gamma^b_t(x_t), \gamma^b_t(y_t) \xrightarrow[b_{out}(e_t)]{b_{in}(e_t)} \gamma^b_{t+1}(x_t), \gamma^b_{t+1}(y_t) \tag{3.42}$$

and r_t^a (resp. r_t^b) is a rule of $\mathcal{A}$ (resp. $\mathcal{B}$). Hence, we can extract an execution E_a (resp. E_b) of $\mathcal{A}$ (resp. $\mathcal{B}$) with schedule S, input history H_{in}^a (resp. H_{in}^b), and output history H_{out}^a (resp. H_{out}^b). Since $\mathbb{C}$ is an acceptable context, E_a and E_b are both $\mathbb{C}$-legal executions. Hence, $H_{out}^a \in Beh(\mathcal{A})(G, H_{in}^a)$ and $H_{out}^b \in Beh(\mathcal{B})(G, H_{in}^b)$. Whence $H_{out} \in \mathcal{C}(G, H_{in})$.

(serial). Let $H_{out} = (S, a_{out})$ be an output history of C corresponding to the input history $H_{in} = (S, b_{in})$. We have to prove that there exists a history $H = (S, b_{out}) \in Beh(\mathcal{B})(G, H_{in})$ such that $H_{out} \in Beh(\mathcal{A})(G, H)$.

By definition, there exists a $\mathbb{C}$-legal execution E of $\mathcal{A} \otimes \mathcal{B}$ with the schedule S, input history H_{in} and output history H_{out}. The execution E can be written

$$\dots (\gamma_t^a, \gamma_t^b) \xrightarrow{\sigma_t} (\gamma_{t+1}^a, \gamma_{t+1}^b) \dots \tag{3.43}$$

where γ_*^a (resp. γ_*^b) are configurations of $\mathcal{A}$ (resp. $\mathcal{B}$), and $\sigma_t = (e_t = (x_t, y_t), r_t)$ is an action such that the rule r_t of $\mathcal{A} \otimes \mathcal{B}$ is equivalent to

$$r_t^a : \ \gamma_t^a(x_t), \gamma_t^a(y_t) \xrightarrow[a_{out}(e_t)]{v_t} \gamma_{t+1}^a(x_t), \gamma_{t+1}^a(y_t) \tag{3.44}$$

$$r_t^b : \ \gamma_t^b(x_t), \gamma_t^b(y_t) \xrightarrow[v_t]{b_{in}(e_t)} \gamma_{t+1}^b(x_t), \gamma_{t+1}^b(y_t) \tag{3.45}$$

for some v_t; and r_t^a (resp. r_t^b) is a rule of $\mathcal{A}$ (resp. $\mathcal{B}$). Then, we define $H = (S, b_{out})$ the history such that $b_{out}(e_t) = v_t$. We have thus constructed an execution E_a (resp. E_b) with schedule S, input history H (resp. H_{in}) and output history H_{out} (resp. H). Since $\mathbb{C}$ is acceptable, E_a and E_b are $\mathbb{C}$-legal executions. Thus, $H_{out} \in Beh(\mathcal{A})(G, H)$ and $H \in Beh(\mathcal{B})(G, H_{in})$.

(feedback). Assume the alphabets of $\mathcal{A}$ are $In(\mathcal{A}) = U \times I$ and $Out(\mathcal{A}) = U \times O$. The protocol $\mathcal{C}$ has input alphabet I, and output alphabet O. Let $H_{out} \in Beh(\mathcal{C})(G, H_{in})$, and write $H_{out} = (S, h_{out})$ (with values in O) and $H_{in} = (S, h_{in})$ (with values in I). We prove that there exists a history $H = (S, h)$ with values in U such that $(S, (h_{out}, h)) \in Beh(\mathcal{A})(G, (S, (h_{in}, h)))$.

By definition, there exists a $\mathbb{C}$-legal execution E of $\mathcal{A}$ with the schedule S, input history H_{in} and output history H_{out}. The execution E can be written

$$\dots \gamma_t \xrightarrow{\sigma_t} \gamma_{t+1} \dots \tag{3.46}$$

where γ_* are configurations of $\mathcal{A}$, and $\sigma_t = (e_t = (x_t, y_t), r_t)$ is an action such that the rule r_t of $\mathcal{C}$ is equivalent to

$$r'_t : \gamma_t(x_t), \gamma_t(y_t) \xrightarrow[(h_{out}(e_t), v_t)]{(h_{in}(e_t), v_t)} \gamma_{t+1}(x_t), \gamma_{t+1}(y_t) \tag{3.47}$$

$$\tag{3.48}$$

for some v_t; r'_t is a rule of $\mathcal{A}$. We simply define $H = (S, h)$ with $h(e_t) = v_t$. Thus, we have constructed an execution E' with schedule S, input history $(S, (h_{in}, h))$, and output history $(S, (h_{out}, h))$. Since $\mathbb{C}$ is acceptable, E' is a $\mathbb{C}$-legal execution. □

3.5 Implementation, Comparison of Behaviours

The notion of behaviour is general enough to represent a problem. For instance, the leader election problem can be seen as the behaviour which associates with

every schedule on some graph, a history with values in $\{0, 1\}$ that eventually permanently outputs 1 at a unique agent. We first define what it means to solve a problem in the model of population protocols, and then define how to assess the relative difficulty of a problem.

Implementation

Roughly speaking, a protocol implements a behaviour, or solves a problem, when all the legal executions satisfy the specifications described by the behaviour or the problem.

The protocol $\mathcal{A}$ is an *implementation* (or a *solution*) of the behaviour B *in the context* $\mathbb{C}$, when $Beh(\mathcal{A}, \mathbb{C})$ is a sub-behaviour of B. In other words, it means that, for any $\mathbb{C}$-legal execution of $\mathcal{A}$ on some graph G, and with input history H_{in}, the corresponding output history H_{out} satisfies $H_{out} \in B(G, H_{in})$.

We will see that very often, depending on the context, the behaviour B_1 admits no implementation by population protocols. However, if we allow the protocol to use another behaviour B_2 (as an oracle), it is possible to implement the original behaviour B_1. We thus introduce the definition of an *implementation of* B_1 *using the behaviour* B_2.

Formally, the protocol $\mathcal{A}$ is an *implementation* (or a *solution*) of B_1 using B_2 in the context $\mathbb{C}$ when there exists a composition C involving B_2 and $Beh(\mathcal{A}, \mathbb{C})$ such that C is a sub-behaviour of B_1.

In some sense, this last definition shows that the behaviour B_1 is weaker than the behaviour B_2. We formalize this notion in the following.

Comparison

In complexity theory, to assess the power of deterministic polynomial-time Turing machine, it is common to study the possibility of a reduction (known as Karp, or Cook reductions) of a problem P_1 to another problem P_2 via some polynomial-time (deterministic) algorithm. If such a reduction exists, P_2 is said to be stronger than P_1, since any solution to P_2 is transformed into a solution to P_1; the transformation being a polynomial-time algorithm. In other words, the problems are compared on the basis of polynomial-time algorithms. Note that, it does not mean that P_1 or P_2 admits a solution which is polynomial-time. However, it implies that if P_2 admits a polynomial-time solution, then P_1 admits a polynomial-time solution. It is also possible to define reductions via other kinds of algorithms, like linear-time, or logarithmic-space, etc.

We mimic this approach in the case of population protocols. Let B_1 and B_2 be two behaviours with the same domain of graphs $\mathcal{F} = Dom(B_1) = Dom(B_2)$. Assume we have a set $\mathcal{H}$ of behaviours, each with the domain $\mathcal{F}$. The behaviour B_1 is *weaker* than the behaviour B_2 *over* $\mathcal{H}$ if there exists a composition C involving one instance of B_2 and (any number of) instances of behaviours from $\mathcal{H}$ such that C is a sub-behaviour of B_1. Roughly speaking, this means that it is possible to compose one instance of B_2 with behaviours of $\mathcal{H}$ to obtain a sub-behaviour of B_1.

Since we deal with population protocols, in this thesis, the family $\mathcal{H}$ will consists of behaviours associated with population protocols for a given context $\mathbb{C}$. By Prop. 2, assuming that $\mathbb{C}$ is acceptable, composing behaviours of pro-

tocols amounts to focus on the behaviour of the corresponding composition of protocols.

In this case, the comparison relation defined above translates to the following: B_1 is weaker than B_2 in the context $\mathbb{C}$, denoted by $B_1 \preccurlyeq_{\mathbb{C}} B_2$, if there exists a population protocol $\mathcal{A}$ and a composition C involving one instance of $Beh(\mathcal{A}, \mathbb{C})$ and B_2, such that C is a sub-behaviour of B_1. The following propositions derives directly from the definitions.

Proposition 3. *The relation $\preccurlyeq_{\mathbb{C}}$ is a preorder.*

Corollary 1. *The relation*

$$B_1 \simeq_{\mathbb{C}} B_2 \underset{def}{\iff} B_1 \preccurlyeq_{\mathbb{C}} B_2 \wedge B_2 \preccurlyeq_{\mathbb{C}} B_1 \tag{3.49}$$

is an equivalence relation.

Note that, like Karp reductions, the reduction of B_1 to B_2 via population protocols does not imply that B_1 or B_2 admit implementations by population protocols. However, it implies that, if B_2 admits an implementation, then so does B_1. This comparison relation is compatible with the notion of sub-behaviour.

Proposition 4. *In any context $\mathbb{C}$, if B_2 is a sub-behaviour of B_1, then $B_1 \preccurlyeq_{\mathbb{C}} B_2$.*

Proof. The *identity* behaviour Id_X, with the same input and output alphabet X, is the behaviour defined as follows: $H_{out} \in P(G, H_{in})$ if and only if $H_{out} = H_{in}$. It is straightforward to see that for any behaviour B with output alphabet X (resp. input alphabet X), we have $Id_X \circ B = B$ (resp. $B \circ Id_X = B$).

Actually, $Id_X = Beh(\mathcal{A}_X)$ where $\mathcal{A}_X$ is the following protocol

$$\begin{aligned} In(\mathcal{A}) = Out(\mathcal{A}) &= X \\ States(\mathcal{A}) &= \{0\} \\ 0,0 &\xrightarrow[o_1,o_2]{i_1,i_2} 0,0 \end{aligned}$$

Therefore, if B_2 is a sub-behaviour of B_1, then the composition $B_2 \circ Beh(\mathcal{A}_{In(B_1)}) = B_2$ is a sub-behaviour of B_1. Whence, $B_1 \preccurlyeq_{\mathbb{C}} B_2$. □

3.6 Related Work

Population Protocol Model

The model described above substantially differs from the original population protocol model. Indeed, in [7], a population protocol simply consists in a set of states, and transition rules without input/output values:

$$p, q \to p', q' \tag{3.50}$$

The aim of the authors was to define the class of predicates on initial configurations of executions that could be "computed" by a network of mobile agents. For instance, the initial states of an agent denote "healthy bird" and "ill bird",

the protocol ensures that eventually every agent in the system has the answer to the question, e.g., "Is there a majority of healthy birds ?". Hence, the "input" to the protocol is encoded in the initial configuration, and the "output" of the protocol is the eventual (and stable) "all zero" or "all one" vector of answers.

Encoding the input in the initial configuration is a rather blunt assumption since in real systems the inputs to the agents are not synchronized, and may vary. In [6], the authors tackle this issue by allowing the inputs to the agents to vary before stabilizing. Hence, although one may still encode initial data in the initial configuration, the model is modified so that the rules of the protocol are provided with input values.

$$p, q \xrightarrow{i,j} p', q' \tag{3.51}$$

Moreover, by using a part of the state from one protocol as an input in the rule of another protocol, this modification allows to compose (serially) population protocols.

Note, however, that in [6], the input values are given as a sequence $(\alpha_t)_{t\in\mathbb{N}}$ of assignments, i.e., maps associating a value with each agent. We take the dual view: the input values are given during transition between configurations.

Finally, the output of an agent is usually encoded as a part of its state. In our case, for sake of symmetry, the output values are produced during transition. The difference is analogous to the difference between Moore machines and Mealey machines.

Oracles, Failure Detectors

The failure detectors were first introduced in [32, 31] to circumvent the impossibility of consensus in asynchronous message-passing systems prone to crash failures [47]. The idea was to augment the system with a blackbox that would give (unreliable) information about the failure pattern. In the original formulation, this information consists in a list of processes identifiers, but one can easily imagine other kinds of information, e.g., numbers of crashed processes, etc.

A definition of *failure detector transformation* was also introduced [31]: a failure detector FD_1 is weaker than a failure detector FD_2 when there exists an (asynchronous) algorithm that uses the output of the oracle FD_2 produce an output matching the specifications of FD_1.

The idea of failure detectors can be translated in the population protocol model via the notion of behaviour. Indeed, a behaviour whose input alphabet reduces to a singleton is exactly a kind of failure detector that gives (possibly unreliable) information about the schedule of events occurring in the system. In a sense, such behaviours are "schedule observers".

The idea of transformation can also be restated: a "schedule observer" O_1 is weaker than a "schedule observer" O_2 when there exists a a composition involving O_2 and behaviours of population protocols, which is a sub-behaviour of O_1. The structure theorems and the fact that the "schedule observers" have no input (i.e. the input alphabet is a singleton) imply that any composition involving O_2 and population protocols amounts to design a population protocol that uses the output of O_2 to produce an output matching the specifications of O_1.

In [46], the authors introduce a new kind of oracle, namely $\Omega?$, that does not only observe the schedule, but also a bit in the states of the agents. The original definition is quite informal, as well as its combination with population protocols. The concept of behaviour allows to formally define such an oracle (see Chap. 6, Sec. 6.4). The main difference with the failure detectors above is that the input alphabet is no more trivial (a singleton). In particular, this fact allows the feedback operation: the population protocol uses the output of the oracle, and the oracle uses the output of the protocol.

We see that the notion of behaviour is general enough to model both oracles observing only the schedules, and more complex oracles allowing feedback operations with population protocols. Actually, the notion of behaviour is general enough to represent a problem. In the same way that Karp reductions are transformations of problems via polynomial-time algorithm, in our case, we look at transformations of behaviours via population protocols.

Chapter 4

Data Collection

4.1 Introduction

In population protocols, the mobile agents may be viewed as moving in a non-deterministic asynchronous way with pairs of agents repeatedly coming close enough to communicate. The choice of the meetings (the mobility of agents) is modeled by the considered fairness condition. The original fairness introduced for population protocols [7] is the global fairness (see Chap. 3, Sec. 3.2), which basically states that, during a fair execution, if any transition between two global configurations γ and γ' is possible infinitely often, then γ' is reached infinitely often during the execution.

On the one hand, this condition is strong, because it relates the possible schedules with the considered protocol. The motivation for such a strong fairness comes from the point of view of population protocols as a model of computation. The computability results of [7] greatly depend on this assumption. On the other hand, this fairness condition gives no easy analytical means to evaluate the convergence times of population protocols.

To achieve this goal, in this chapter, we adopt the fairness with cover times (see Chap. 3, Sec. 3.2) introduced in [16]. Recall that, first, it deals only with meetings between agents, i.e., it has no knowledge about local states and/or protocol transitions. Second, it provides a notion of synchrony. The cover time of an agent x is the minimum number of events happening in the system for being certain that x has met every other agent (directly). Such a condition imposes that one cannot postpone some meeting arbitrarily often, as it is possible in [7]. Actually, the cover time property may be viewed as an introduction of "partial synchrony" assumptions [45]; partial, because the cover times are not assumed to be known to the agents. The main advantage of these differences is that they allow to compute deterministic time complexities (or event complexities), expressed in the number of events.

The assumption that an agent communicates with all other agents periodically, with a bounded period, has been experimentally justified for some types of mobility. Indeed, in the case of human or animal mobility within a bounded area or with a "home coming" tendency (the tendency to return to some specific places periodically), the statistical analysis of experimental data sets confirms this assumption (e.g., [51, 54, 25]). These data sets concern students on a campus [1], participants to a network conference [29] or visitors at

Disneyland. All exhibit the fact that the *inter-contact time* (ICT) between two agents, considered as a random variable, follows a truncated Pareto distribution. In particular, this involves that the ICTs, measured in terms of a real time, are finite in practice. Thus, they are also finite when measured in events. So is the cover time of an agent, which is the maximum of its ICTs measured in events.

This chapter presents, on examples, some techniques for computing the event complexity of population protocols. For this purpose, we propose and analyze some adapted versions of the existing *data collection* protocol, used by the ZebraNet project [53]. ZebraNet is a project conducted by the Princeton University and deployed in central Kenya. It aims at studying populations of zebras using sensors attached to the animals. This project developed an history-based protocol to deliver the sensed values to the base station. When an agent x has to deliver its data, it may relay it to an agent y that has recently met the base station more frequently. The protocol assumes that y will continue meeting the base station frequently in the near future and will deliver the data sooner.

We incorporate a "one-shot" version (executing the data delivery only once) of this ZebraNet protocol in the population protocol model with cover times and we study analytically the complexity of the resulting protocol, as well as other variants. For the sake of simplicity and due to the constraints of the model (e.g., pairwise instead of multiwise interactions, finite cover times for all agents), the resulting protocols are only simplified versions of the original one.

The scope of this work is worst case analysis. However, it is important to note that an average case stochastic analysis is necessary to more accurately compare data collection protocols. Still, the given worst case analysis introduces several techniques that may prove useful in future studies of both average and worst case time analysis. Moreover, in order to understand why some protocols have a better average complexity, it is possible to consider and analyze some specific cases of executions. We give examples of such cases in Sec. 4.6. Refer to this section also for additional discussion and protocols' comparisons.

[16] is the most relevant work to the one presented here. There, several data collection protocols are proposed and their worst case complexity analysis is presented, in the model of population protocols with cover times. In addition, a lower bound for the worst case convergence time for any data collection task is proved and one of the proposed protocols is proved to be optimal in terms of this bound (its complexity is less than $2 \cdot cv_{min}$). However, in contrast with the current work, the communication model of [16] is stronger than the one we assume here. Namely, [16] assumes that two interacting agents are able to compare their cover times accurately. Thus, it is somewhat difficult to compare the protocols proposed here (which do not rely on any knowledge of cover times) with those in [16]. Nevertheless, the time complexity gap between the protocols in [16] and the protocols presented here should not be so large when considering average complexity. It can be easily justified by the complexity analysis of some prevalent execution scenarios for specific cases of populations (see Sec. 4.6).

This work has been published in [13].

The Problem

Let G be a complete communication graph with n agents; n being unknown to the agents. The Base Station (BS) is a special distinguishable agent with extended resources. BS is required here only by the nature of the data collection problem. In contrast with BS, all the other agents are finite-state, anonymous and are referred in the chapter as *mobile*. We denote by G^* the set of mobile agents. Mobile agents are enumerated from 1 to $n-1$.

The *data collection* problem is defined as follows. Each mobile agent initially owns an *input (data) value* — the value provided by the sensor (e.g., temperature or heart-rate). Each input value has to be *delivered* to BS exactly once. When this happens, we say that a *legal* configuration is reached. An execution is said to *converge*, if it reaches a legal configuration. A protocol is said to converge, if all its executions converge. The *length* of an execution that converges is the minimum number of events until convergence. The *worst case event complexity* of a protocol is the maximum length of its executions.

Context and Notations

The ZebraNet Protocol, and the variants proposed below, aim at solving the data collection problem. These protocols are studied in the model with the *cover time property* over the family of all *complete* communication graphs. There are no failures (no crashes, nor transient faults). In particular, every agent is initialized with its input value at the beginning of every execution.

As already explained in Chap. 3, Sec. 3.2, each agent x is associated with a positive integer cv_x, called the *cover time* of x. Agents are not assumed to know the cover times. We denote by $\overline{cv}$ the vector of agents' cover times and by cv_{min} (resp. cv_{max}) the minimum (resp. maximum) cover time in $\overline{cv}$.

A *fastest* (resp. *slowest*) agent x has $cv_x = cv_{min}$ (resp. $cv_x = cv_{max}$). We say that the cover time vector $\overline{cv}$ is *uniform* if all its entries are equal, i.e., $cv_{min} = cv_{max}$. In this case, we denote by cv the common value of the agents' cover times.

It is assumed that, during a meeting, an agent x can transfer a set of values to another agent y; it is also assumed that doing so, the agent x *does not keep any copy* of the transferred values. For a meeting event $(x\ y)$, the notation $(x\ \underline{y})$ indicates a *transfer* of values from x to y. However, the notation $(x\ y)$ does not imply that there is no transfer. To specify one of the values being transferred, v for example, we note $(x\ \underline{y})^{(v)}$.

In this chapter, given any finite schedule S and any positive integer l, the schedule S^l denotes the schedule obtained by repeating l times the sequence S. The infinite schedule S^ω denotes the infinite repetition of S.

Overview

The chapter is divided as follows. In Sec. 4.2, it is shown that some execution of the original ZebraNet protocol does not converge. To circumvent this result, two variants are proposed, MZP1 (Sec. 4.3) and MZP2 (Sec. 4.4), and their worst-case complexity is analyzed. For sake of simplicity, in MZP1 and MZP2, it is assumed that every agent can hold as many values as there are in the

system (unbounded memory). In Sec 4.5, versions assuming bounded memory are presented and analyzed.

4.2 Non-convergence of ZebraNet

In the original ZebraNet data collection protocol [53], an agent chooses, among the agents in its range, the one which is the most likely to meet BS in a near future, and transfers its values to it. In population protocols, agents interact only in pairs, in contrast to the multiwise communications possible in ZebraNet. Hence, the ZebraNet Protocol (ZP), Algorithm 1 presented below, is a restricted version of the original ZebraNet protocol. However, as any execution of ZP is also an execution of the original protocol, the non convergence of ZP involves the non convergence of the latter.

In ZP, the state of an agent x is defined by the integer variables $accumulation_x$ and $distance_x$, a set of data values $values_x$ (the type of which we do not specify) and an integer constant $decay$. The value of $decay$ is predefined and is the same for every agent. The integer variables are initially set to 0. The set $values_x$ initially holds the input value of agent x.

For sake of simplicity, we assume first that the memory available to each agent is large enough, so that it can store the values of all other agents. This assumption prevents memory overflows during transfers. In other words, and as already noted in the introduction, we assume first that agents have $O(n)$ memory size. The case of bounded (constant) memory is analyzed in Sec. 4.5.

In Algorithm 1, when an agent x meets BS, its variable $accumulation_x$ is incremented and $distance_x$ is reset to 0. When an agent x meets another mobile agent, its variable $distance_x$ is incremented. If $distance_x$ becomes larger than $decay$, $accumulation_x$ is decremented and $distance_x$ is reset to 0.

When an agent x holds some values in $values_x$ and meets another mobile agent y, if $accumulation_y$ is strictly greater than $accumulation_x$, then agent x transfers all its values to agent y. An agent always transfers all its values when it meets BS.

It appears that some executions of ZP do not converge. Indeed, a value can circulate between mobile agents without ever being delivered to BS.

Proposition 5 (Non Convergence of ZP)**.** *For any graph G of $n \geq 4$ agents, for any $decay \geq 1$, there exists a uniform cover time vector $\overline{cv}$ for which there is an execution of* ZP *that does not converge.*

Proof. Consider a graph G of $n \geq 4$ agents and a constant $decay \geq 1$. We first define specific sequences of events :

- $U_1 = (1\ BS)(2\ 1)$
- $V = [(2\ 3)\dots(2\ n-1)] \cdot [(3\ 4)\dots(3\ n-1)] \cdot \dots \cdot (n-2\ n-1)$
  All mobile agents, except agent 1, meet each other once.
- $W_1 = (1\ 2)\dots(1\ n-1)$
  Agent 1 meets every other mobile agent once.
- $U_2 = (2\ BS)(1\ 2)$

Algorithm 1: ZebraNet Protocol

```
when x meets BS do
    <x transfers values_x to BS>
    accumulation_x := accumulation_x + 1
    distance_x := 0
end when
when x meets y ≠ BS do
    if accumulation_x < accumulation_y ∧ <values_x is not empty> then
        <x transfers values_x to y>
    end if
    distance_x := distance_x + 1
    if distance_x > decay then
        if accumulation_x ≠ 0 then
            accumulation_x := accumulation_x − 1
        end if
        distance_x := 0
    end if
end when
```

- $W_2 = (2\ 1)(2\ 3)\ldots(2\ n-1)$
  Agent 2 meets every other mobile agent once.
- $Z = (3\ BS)\ldots(n-1\ BS)$
  All mobile agents, except agents 1 and 2, meet BS.

We choose an integer g such that $g \cdot (n-3) \geq decay + 1$.
Now we build a schedule $\mathcal{S}$ as follows :

$$X = U_1\ V^g\ W_1^g\ U_2\ W_2^g\ Z$$
$$\mathcal{S} = X^\omega$$

By construction, in X, all agents meet each other at least once. For any mobile agent x, we choose $cv_x = cv = |X|$. That implies that $\mathcal{S}$ satisfies the cover time property. Precisely, $cv = g \cdot \frac{(n-3)(n-2)}{2} + (2g+1)(n-2) + 3$.

We claim that the input value v of agent 2 is never delivered to BS. To see that, consider what happens when the sequence X is applied to an initial configuration γ_0. During $U_1 = (1\ BS)(1\ 2)$, agent 1 receives the input value v of agent 2. During the sequence V^g, only agents 2 to $n-1$ are involved, thus, at the end of V^g, agent 1 still holds v. Then, comes the sequence W_1^g, where agent 1 meets every other mobile agent g times. Since agents 2 to $n-1$ have not met BS yet, their variables *accumulation* equal 0 and agent 1 cannot transfer v to any of them. In addition, since agent 1 is involved in $g \cdot (n-2) \geq decay + 1$ meetings (thanks to the choice of g), the decay mechanism of ZP implies that at the end of W_1^g, the variable $accumulation_1$ of agent 1 equals 0.

Therefore, during $U_2 = (2\ BS)(2\ 1)$, agent 1 transfers v to agent 2. In W_2^g, agent 2 is involved in $g \cdot (n-2) \geq decay + 1$ meetings with other mobile agents. Since all their variables *accumulation* equal 0, agent 2 keeps v. Note that the decay mechanism implies that at the end of W_2^g, the variable $accumulation_2$ of agent 2 equals 0. Finally, during Z, all mobile agents $x \notin \{1, 2\}$ meet BS and

increment their variable $accumulation_x$ accordingly. Therefore, the application of the sequence X to an initial configuration γ_0 leads to a configuration γ_1 that satisfies the following property $\mathcal{P}$:

- agent 2 holds its input value v
- $accumulation_1 = accumulation_2 = 0$
- $\forall x \in G^* \setminus \{1, 2\}, accumulation_x = 1$

Now, apply X to γ_1. At the end of U_1, agent 1 has received v from agent 2 and satisfies $accumulation_1 = 1$. During V^g, each mobile agent $x \neq 1$ is involved in $g \cdot (n-3) \geq decay + 1$ meetings. Therefore, thanks to the decay mechanism, at the end of V^g, all agents, except agent 1, have their variable $accumulation$ set to 0. Hence, during W_1^g, agent 1 cannot transfer v to any other mobile agents. In addition, the decay mechanism implies that at the end of W_1^g, the variable $accumulation_1$ of agent 1 equals 0. Then, we can apply the same arguments as in the previous paragraph to the sequence $U_2\ W_2^g\ Z$ that follows. Thus, the application of the sequence X to γ_1 leads to a configuration γ_2 that also satisfies the property $\mathcal{P}$.

Hence, no matter how many sequences X are applied, the input value v of agent 2 is never delivered to BS. □

4.3 Modified ZebraNet Protocol 1

To obtain convergence, the algorithm is modified by ensuring that a mobile agent that transfers data to another mobile agent can no longer accept data. For this purpose, we add a boolean variable $active_x$, initially set to *true*, that indicates whether agent x is *active* or not, and we impose that only active agents can receive values. Once an active agent has transferred its values to another *mobile* agent, it becomes *inactive*. Algorithm 2 below presents the pseudo-code of MZP1.

Convergence

We now show that any execution of MZP1 converges. The proof relies on the fact that the set of active agents cannot increase, so that at some point of any execution, it remains constant. From that point, there is no value transferred between the mobile agents, and since all mobile agents eventually meet BS (due to the cover time property), all values are eventually delivered.

Proposition 6 (Convergence of MZP1)**.** MZP1 *converges.*

Proof. Let $\mathcal{E}$ be an execution. We denote by $ACT(k)$ the set of active agents in the k-th configuration in $\mathcal{E}$. The sequence $(ACT(1), ACT(2), \dots)$ is non-increasing, thus it is eventually constant : $\exists k_0 \in \mathbb{N}, \forall k \geq k_0, ACT(k) = ACT(k_0)$. Starting from the k_0-th configuration, there cannot be any further transfer between two active agents. Otherwise, the set of active agents would decrease. Also, according to Algorithm 2, there cannot be any transfer from an active agent to another inactive agent, nor from an inactive agent to an inactive agent. In other words, once the set of active agents remains constant, there cannot be any transfer between two mobile agents. Since all

Algorithm 2: Modified ZebraNet Protocol 1

```
when x meets BS do
    <x transfers values_x to BS>
    accumulation_x := accumulation_x + 1
    distance_x := 0
end when
when x meets y ≠ BS do
    if accumulation_x < accumulation_y ∧ active_y ∧ <values_x is not empty>
    then
        <x transfers values_x to y>
        active_x := false
    end if
    distance_x := distance_x + 1
    if distance_x > decay then
        if accumulation_x ≠ 0 then
            accumulation_x := accumulation_x − 1
        end if
        distance_x := 0
    end if
end when
```

mobile agents meet BS in the next cv_{max} events, all the values are eventually delivered. □

Upper Bound to the MZP1 Complexity

In this section, we present an upper bound to the number of events needed to collect all the values at the base station. First we define the notion of *path followed by a value*.

Definition 2 (Path followed by a value)**.** *Let $\mathcal{E}$ be an execution and v be an initial value of some agent. The path followed by v in $\mathcal{E}$ is the sequence (possibly infinite) of the mobile agents that successively carry v.*

For example, let x_1 be an agent whose input value is v. It is possible that x_1 transfers v to some agent x_2, then agent x_2 transfers v to some agent x_3, which finally delivers v to BS. In this case, the path followed by v is $x_1x_2x_3$. Note that, without the *active* variable (e.g. in ZP), the agent x_3 could be the agent x_1.

Proposition 7 (Upper Bound - MZP1)**.** *For any graph G of $n \geq 3$ agents, for any cover time vector $\overline{cv}$, and for any decay ≥ 1, any execution of* MZP1 *converges in no more than $\sum_{x \in A^*} cv_x - 2 \cdot (n-2)$ events.*

Proof. Let $\mathcal{E}$ be an execution of MZP1. By Prop. 6, $\mathcal{E}$ converges, i.e., all the values are eventually delivered. Let v be an input value of some agent x_1 such that v is the last delivered value in $\mathcal{E}$. Consider the path π followed by v in $\mathcal{E}$. It is of the form $x_1x_2 \ldots x_k$ for some $k \geq 1$, x_k being the agent that delivers v to BS. Since a mobile agent becomes inactive as soon as it transfers some values,

all the agents appearing in π are different. Hence, we have $1 \leq k \leq n-1$. Then, the execution $\mathcal{E}$ can be written as the following sequence of events[1]:

$$\mathcal{E} = \underbrace{\left[\dots(x_1\ \underline{x_2})^{(v)}\right]}_{e_1}\ \underbrace{\left[\dots(x_2\ \underline{x_3})^{(v)}\right]}_{e_2}\ \dots\ \underbrace{\left[\dots(x_{k-1}\ \underline{x_k})^{(v)}\right]}_{e_{k-1}}\ \underbrace{\left[\dots(x_k\ \underline{BS})^{(v)}\right]}_{e_k}\ \dots$$

The subsequence e_i starts after the transfer of v from x_{i-1} to x_i and ends with the transfer of v from x_i to x_{i+1}, except e_1 (that starts with the beginning of $\mathcal{E}$) and e_k (that ends when v is delivered to BS).

Now, we show that for every $2 \leq i \leq k-1$, the length of e_i is upper bounded by $cv_{x_i} - 2$. Consider i in this range and the following sequence of events in $\mathcal{E}$, $e'_i := \left[(x_{i-1}\ \underline{x_i})^{(v)} \dots (x_i\ \underline{x_{i+1}})^{(v)}\right]$. Note that x_i does not meet BS during e'_i. Hence, $|e'_i| \leq cv_{x_i} - 1$ and $|e_i| \leq cv_{x_i} - 2$. For the same reason, $|e_1| \leq cv_{x_1} - 1$. For $i = k$, as before (for e'_i), starting with event $(x_{k-1}\ \underline{x_k})^{(v)}$ and till the last event in e_k, x_k does not meet BS. Only during this last event in e_k, x_k necessarily meets BS and finally delivers v. Hence, $|e_k| \leq cv_{x_k} - 1$. Therefore, the value v is delivered to BS in no more than

$$\begin{aligned} |e_1| + \dots + |e_k| &\leq (cv_{x_1} - 1) + (cv_{x_2} - 2) + \dots + (cv_{x_{k-1}} - 2) + (cv_{x_k} - 1) \\ &\leq \underbrace{\sum_{x \in \pi} cv_x - 2 \cdot (|\pi| - 1)}_{T} \end{aligned}$$

Now, we denote by $\alpha_1 > \dots > \alpha_r$ the distinct values of the cover times of the mobile agents. Note that $\alpha_r \geq n-1 \geq 2$. We denote by Γ_α the number of mobile agents in the system with a cover time equal to α, and by π_α the number of agents in π (there are only mobile agents in π, by construction) with a cover time equal to α. Hence, $|\pi| = \pi_{\alpha_1} + \dots + \pi_{\alpha_r}$ and $n-1 = \Gamma_{\alpha_1} + \dots + \Gamma_{\alpha_r}$. Then, we have $T = \pi_{\alpha_1} \cdot \alpha_1 + \dots + \pi_{\alpha_r} \cdot \alpha_r - 2 \cdot (|\pi| - 1)$. By replacing π_{α_r} with $|\pi| - \pi_{\alpha_1} - \dots - \pi_{\alpha_{r-1}}$, we get :

$$\begin{aligned} T &= \pi_{\alpha_1} \cdot (\alpha_1 - \alpha_r) + \dots + \pi_{\alpha_{r-1}} \cdot (\alpha_{r-1} - \alpha_r) + |\pi| \cdot (\alpha_r - 2) + 2 \\ &\leq \Gamma_{\alpha_1} \cdot (\alpha_1 - \alpha_r) + \dots + \Gamma_{\alpha_{r-1}} \cdot (\alpha_{r-1} - \alpha_r) + (n-1) \cdot (\alpha_r - 2) + 2 \\ &\leq \Gamma_{\alpha_1} \cdot \alpha_1 + \dots + \Gamma_{\alpha_r} \cdot \alpha_r - 2 \cdot (n-2) \\ &\leq \left(\sum_{x \in G^*} cv_x\right) - 2 \cdot (n-2) \end{aligned}$$

Since all the other values are delivered before v, $\mathcal{E}$ converges in no more than $\sum_{x \in G^*} cv_x - 2 \cdot (n-2)$ events. □

Lower Bound to the MZP1 Complexity

Now, we show that the upper bound stated in Prop. 7 is optimal. Building a "long" worst case execution is made difficult by two contradictory constraints. On the one hand, the mechanism of *accumulation* variables and of *decay*, in particular when the value of the constant *decay* is small, forces to add events

[1] We remind the reader that this is an abusive notation, refer to Sec. 4.1.

in the construction for controlling the transfers. However, on the other hand, the cover time property forces some specific events (and not necessarily those needed for the construction) to happen before fixed deadlines (given by the cover times). For the sake of clarity, we assume a uniform cover time vector $\overline{cv}$, for which the upper bound stated in Prop. 7 becomes $(n-1)\cdot cv - 2\cdot(n-2)$. In the sequel, we build an execution that converges in exactly $(n-1)\cdot cv - 2\cdot(n-2)$ events and satisfies the cover time property for $\overline{cv}$.

Proposition 8 (Lower Bound - MZP1)**.** *For any graph G of $n \geq 4$ agents, for any decay ≥ 1, there exists a uniform cover time vector $\overline{cv}$ for which there is an execution of* MZP1 *that converges in exactly $(n-1)\cdot cv - 2\cdot(n-2)$ events.*

Proof. We consider a graph G of $n \geq 4$ agents and a constant $decay \geq 1$. Let g be an integer such that $g\cdot(n-3) \geq decay + 1$. We consider a uniform cover time vector $\overline{cv}$, the value of which is defined later.

We build, step by step, an execution in which the input value of agent 1 is successively carried by every other agent. First, for each $1 \leq k \leq n-2$, we consider a sequence E_k of length cv in which the value v is transferred from agent k to $k+1$, and another sequence Δ in which agent $n-1$ delivers v to BS. Since a schedule is an infinite sequence, we complete by repeating a pattern Ω and we define $\mathcal{S} = E_1E_2\cdots E_{n-2}\Delta\Omega^\omega$. The difficulty lies in the definition of the sequences E_k, Δ and Ω for the schedule $\mathcal{S}$ to satisfy the cover time property and for the value v to be delivered to BS at the end of Δ.

For this purpose, we define specific sequences as follows :

- For $1 \leq k \leq n-1$, $U(k)$ is a sequence of events in which all the mobile agents, except agent k, meet each other once. Hence, each mobile agent (except agent k) is involved in $n-3$ meetings. We have $|U(k)| = \frac{(n-3)(n-2)}{2}$.
- For $1 \leq k \leq n-1$, $V(k)$ is a sequence in which agent k meets every other mobile agent once. We have $|V(k)| = n-2$.
- For $1 \leq p \leq q \leq n-1$, $B_q^p = (q\ BS)(q-1\ BS)\ldots(p\ BS)$ is a sequence in which each agent x, from q to p, successively meets BS in this order. We have $|B_q^p| = q-p+1$.
- For $1 \leq p \leq q \leq n-2$, $C_q^p = [(q\ q+1)(q\ BS)]\ldots[(p\ p+1)(p\ BS)]$ is a sequence in which each agent x, from q to p, meets its successor $x+1$, then BS. We have $|C_q^p| = 2\cdot(q-p+1)$.

Examine the effect on the executions of the iteration, g times, of $U(k)$ and $V(k)$. In $U(k)^g$, each mobile agent $x \neq k$ is involved in $g\cdot(n-3) \geq decay+1$ meetings with other mobile agents. Thus, thanks to the decay mechanism, applying $U(k)^g$ to any configuration of the system makes each non-zero $accumulation_x$, for each mobile agent $x \neq k$, decrease at least by one. The same argument shows that applying $V(k)^g$ to any configuration makes $accumulation_k$ decrease at least by one, unless $accumulation_k$ already equals 0. In other words, the sequences $U(k)^g$ and $V(k)^g$ help resetting the variables $accumulation$.

Now, consider a configuration in which for all $x \in G^*, accumulation_x = 0$. In addition, assume that some mobile agent k, such that $1 \leq k \leq n-2$, holds a value w and that agent $k+1$ is active (i.e., it can receive values). Then, it is easy to see that during the sequence $B_{n-1}^{k+1} \cdot C_k^1 = B_{n-1}^{k+2}(k+1\ BS)(k\ k+1)(k\ BS)C_{k-1}^1$, agent k transfers w to $k+1$. Moreover, at the end, every $accumulation_x$ (for a mobile agent x) equals 1. In other words, applying $B_{n-1}^{k+1} \cdot C_k^1$ to the appropriate configuration results in a transfer from agent k to agent $k+1$.

We also define, for each $1 \leq k \leq n-2$, a "filling" sequence F_k of meetings between mobile agents. We only require that $|F_k| = n-2-k$ (which implies that $F_{n-2} = \emptyset$). The purpose of the sequence F_k is to ensure that the length of each E_k is constant (independent of k). Now we are ready to define the sequences E_k $(1 \leq k \leq n-2)$, Δ and Ω :

$$E_1 = \underbrace{U(2)^g V(2)^g}_{prologue} \cdot \underbrace{U(1)^g(1\ 2)F_1}_{center} \cdot \underbrace{B_{n-1}^2 C_1^1}_{epilogue}$$

$$(2 \leq k \leq n-2)\ E_k = \underbrace{U(k)^g V(k)^g}_{prologue} \cdot \underbrace{U(k)^g(k\ k+1)F_k}_{center} \cdot \underbrace{B_{n-1}^{k+1} C_k^1}_{epilogue}$$

$$\Delta = U(n-1)^g V(n-1)^g U(n-1)^g \cdot (n-2\ n-1) \cdot (n-1\ BS)$$

$$\Omega = \underbrace{B_{n-1}^{n-1} C_{n-2}^1}_{\text{epilogue of } E_{n-2}} \cdot \Delta = (n-1\ BS)C_{n-2}^1 \cdot \Delta$$

Then, for having the result, we set $cv = |E_k|$. Precisely, we have $cv = g \cdot (n-3)(n-2) + (g+2)(n-2) + 2$.

(Time to convergence). Now, we focus on the circulation of the input value v of agent 1. Let γ_1 be an initial configuration. The *prologue* and the *center* of E_1 only involves meetings between mobile agents, and, since each mobile agent has its variable *accumulation* equal to 0, there is no transfer. At the end of the *epilogue* of E_1, the previous remarks show that agent 1 has transferred v to agent 2 and each mobile agent x satisfies $accumulation_x = 1$. Moreover, during the epilogue of E_1, every mobile agent $x \neq 1$ has transferred its input value to BS. We denote by γ_2 the configuration at the end of E_1.

Consider now the prologue $U(2)^g V(2)^g$ of E_2. At the end of $U(2)^g$, every $accumulation_x$ with $x \neq 2$ is set to 0. Thus, during $V(2)^g$, agent 2 does not transfer v to anyone. In addition, at the end of the prologue of E_2 each mobile agent's *accumulation* variable is set to 0. Hence, during the center of E_2, there is no transfer. It is only during the epilogue of E_2 that agent 2 transfers v to agent 3 (which is still active since it has not transferred any value to any other mobile agent). At the end of E_2, agent 3 holds the value v and every mobile agent x satisfies $accumulation_x = 1$. Therefore, the process can be iterated.

At the end of E_{n-2}, agent $n-1$ holds the value v. Every mobile agent $1, \ldots, n-2$ is inactive since it has transferred v to its successor, and cannot receive v again. Therefore, the value v is delivered to BS exactly at the end of $\Delta = U(n-1)^g V(n-1)^g U(n-1)^g \cdot (n-2\ n-1) \cdot (n-1\ BS)$.

A simple calculation shows that $|\Delta| = cv - 2 \cdot (n-2)$. Hence, with the schedule $\mathcal{S}$, the algorithm converges in exactly $(n-1) \cdot cv - 2 \cdot (n-2)$ events.

$U(2)^g$	$V(2)^g$	$U(1)^g$	(1 2)	F_1	$(n-1\ BS)B^3_{n-2}$	(2 BS)	C^1_1
$U(2)^g$	$V(2)^g$	$U(2)^g$	(2 3)	$F_2(n-1\ BS)$	$B^3_{n-2}(2\ 3)$	(2 BS)	C^1_1

Table 4.1: Comparison of the same relative positions in E_1 and E_2.

($\mathcal{S}$ satisfies the cover time property). To show this, we first introduce a supplementary definition. If e is an event in some E_k, for $1 \leq k \leq n-2$ (resp. Δ), then its *relative position* within E_k (resp. Δ) is defined as follows. If e is the first event in E_k, then its relative position is 1. If it is the second, its relative position is 2, and so on. Tables 4.1, 4.2 and 4.3 compare the same relative positions of different sequences in $\mathcal{S}$. Sequences in the same column start at the same relative position.

We have to check that in any sequence Z of cv consecutive events in $\mathcal{S}$, each agent meets every other agent at least once. Note that if a sequence Z contains (or can be reordered to contain) a prologue and an epilogue (not necessarily from the same sequence E_k), then, in Z, each agent meets every other agent at least once. The following analysis relies on this observation. We denote by Z_1 the first event in Z. In the sequel, we distinguish several cases according to the position of Z_1 in $\mathcal{S}$, and, for each case, several subcases.

- Z_1 is in E_1 (refer to Table 4.1)
  - Z_1 is in $U(2)^gV(2)^g$, in the prologue of E_1.
    The sequences E_1 and E_2 have the same prologue. Thus, in this case, for any event that appears in E_1, from the first event until Z_1 (such event is not in Z), a similar event will appear in E_2 and hence, in Z. Therefore, Z can be reordered to contain the prologue and the epilogue of E_1.
  - Z_1 is in the center of E_1.
    Then, Z contains the epilogue of E_1 and the prologue of E_2.
  - Z_1 is in the epilogue of E_1.
    If Z_1 is the first event of the epilogue of E_1, then Z obviously contains the epilogue of E_1 and the prologue of E_2. If we shift Z_1 to the right by one position, then the sequence Z looses the event $(n-1\ BS)$, but a similar event appears in Z from the center of E_2 (see Table 4.1). This is due to the fact that the sequence $(n-1\ BS)B^3_{n-2}$ of E_1 starts one event later than the same sequence in E_2. We can repeat this argument until the entire sequence $(n-1\ BS)B^3_{n-2}$ of E_1 is "consumed". Hence, if Z_1 is in $(n-1\ BS)B^3_{n-2}$, we can reorder Z in order to contain the epilogue of E_1 and the prologue of E_2. The last subcase is when Z_1 is in $(2\ BS)C^1_1$. This sequence has the same relative position in E_1 and in E_2, thus Z can also be reordered to contain the epilogue of E_1, and the prologue of E_2.

- Z_1 is in E_k $(2 \leq k \leq n-3)$ (refer to Table 4.2)
  - Z_1 is in the prologue $U(k)^gV(k)^g$ of E_k.

$U(k)^g$ $U(k+1)^g$	$V(k)^g$ $V(k+1)^g$	$U(k)^g$ $U(k+1)^g$	$(k\ k+1)$ $(k+1\ k+2)$	F_k $F_{k+1}(n-1\ BS)$	$(n-1\ BS)B_{n-2}^{k+2}$ $B_{n-2}^{k+2}(k+1\ k+2)$	$(k+1\ BS)$ $(k+1\ BS)$	C_k^1 C_k^1

Table 4.2: Comparison of the same relative positions in E_k and E_{k+1} ($2 \leq k \leq n-3$).

If Z_1 is in $U(k)^g$ in the prologue, then, since $U(k)^g$ also appears in the center of E_k, Z can be reordered to contain the prologue and the epilogue of E_k. If Z_1 is in $V(k)^g$ in the prologue , then Z contains the epilogue of E_k. Thus every mobile agent meets the base station. Z also contains $U(k)^g$ from the center of E_k, hence every mobile agent, except agent k, meets each other agent at least once. We just have to check that agent k meets every other mobile agent. Z contains the sequence $U(k+1)^g$ from the prologue of E_{k+1}, in which agent k meets every other mobile agent except agent $k+1$. But Z also contains the event $(k\ k+1)$ from the center of E_k. Hence, every agent meets every other agent at least once.

- Z_1 is in the center of E_k.
  Then, Z contains the epilogue of E_k and the prologue of E_{k+1}.
- Z_1 is in the epilogue of E_k.
  This case is analogous to the case in which Z_1 is in the epilogue of E_1.

- Z_1 is in E_{n-2} (refer to Table 4.3)
  - Z_1 is in the prologue $U(n-2)^g V(n-2)^g$ of $E_n - 2$.
    If Z_1 is in $U(n-2)^g$ in the prologue, then, since $U(n-2)^g$ also appears in the center of E_{n-2}, Z can be reordered to contain the prologue and the epilogue of E_{n-2}. If Z_1 is in $V(n-2)^g$ in the prologue, then Z contains the epilogue of E_{n-2}. Thus, every mobile agent meets the base station. Z also contains $U(n-2)^g$ from the center of E_{n-2}, hence every mobile agent, except agent $n-2$, meets each other at least once. We just have to check that agent $n-2$ meets every other mobile agent. Z contains the sequence $U(n-1)^g$ from Δ, so agent $n-2$ meets every other mobile agent except agent $n-1$. But Z also contains the event $(n-2\ n-1)$ from the center of $E_n - 2$. Hence, every agent meets every other agent at least once.
  - Z_1 is in the center of E_{n-2}.
    Then, Z contains the epilogue of E_{n-2} and the sequence $U(n-1)^g V(n-1)^g$ from Δ. So every agent meets every other agent at least once in Z.
  - Z_1 is in the epilogue of E_{n-2}.
    If Z_1 is the first event of the epilogue, then it is not difficult to see that $Z = \Omega$ and that in Ω, every agent meets each other at least once. By construction, the suffix of the schedule $\mathcal{S}$ consists of an infinite repetition of Ω. Therefore, no matter how many times Z_1 is shifted to the right, Z can always be reordered to be identical to Ω.

$U(n-2)^g$ $U(n-1)^g$	$V(n-2)^g$ $V(n-1)^g$	$U(n-2)^g$ $U(n-1)^g$	$(n-2\ n-1)$ $(n-2\ n-1)$	$F_{n-2}=\emptyset$ $\emptyset$	$(n-1\ BS)$ $(n-1\ BS)$	C^1_{n-2}

Table 4.3: Comparison of the same relative positions in E_{n-2} and Δ.

This last argument shows that the suffix Ω^ω of $\mathcal{S}$ satisfies the cover time property. As a conclusion, in all cases, every agent meets each other at least once in every Z in $\mathcal{S}$. Thus, the schedule $\mathcal{S}$ satisfies the cover time property. □

4.4 Modified ZebraNet Protocol 2

As already explained, the non convergence of ZP is due to the fact that a value can circulate between two or more mobile agents, without ever being delivered to the base station. To prevent that, in MZP1, we imposed that a mobile agent that transfers some values cannot receive any values later. Another way to prevent the cycling of values is to impose that a mobile agent receiving some values cannot transfer them to any other mobile agent later. For this purpose, an *active* bit is also introduced, which yet has not the same functionality as in MZP1. Algorithm 3 below presents the resulting protocol, called MZP2.

Algorithm 3: Modified ZebraNet Protocol 2

```
when x meets BS do
    <x transfers its values to BS>
    accumulation_x := accumulation_x + 1
    distance_x := 0
end when
when x meets y ≠ BS do
    if accumulation_x < accumulation_y ∧ active_x ∧ <values_x is not empty>
    then
        <x transfers its values to y>
        active_y := false  // agent y becomes inactive
    end if
    distance_x := distance_x + 1
    if distance_x > decay then
        if accumulation_x ≠ 0 then
            accumulation_x := accumulation_x − 1
        end if
        distance_x := 0
    end if
end when
```

Upper Bound to the MZP2 Complexity

Proposition 9 (Upper Bound - MZP2)**.** *For any graph G of $n \geq 1$ agents, for any cover time vector $\overline{cv}$ and for any decay ≥ 1, any execution of* MZP2 *converges in no more than $2 \cdot cv_{max} - 2$ events.*

Proof. Consider an execution of MZP2 and an agent x with input value v. During the first cv_x events, there are two possibilities. Either agent x does not transfer v to any other mobile agent, but straightly to BS. In this case, v is delivered in at most cv_x events. Otherwise, x meets some mobile agent y (before it meets BS), in an event $(x\ \underline{y})^{(v)}$, and transfers v to y. This happens in at most cv_x-1 events. According to Algorithm 3, after such a transfer, y becomes inactive. Now, agent y cannot transfer v to any other mobile agent. This implies that agent y will transfer v to BS during the next cv_y events (starting with event $(x\ \underline{y})^{(v)}$). Hence, v is delivered to BS in at most $cv_x + cv_y - 2$ events. In all cases, any value v is delivered to the base station in no more than $2 \cdot cv_{max} - 2$ events. □

Lower Bound to the MZP2 Complexity

Proposition 10 (Lower Bound - MZP2)**.** *For any graph G of $n \geq 4$ agents and any decay ≥ 1, there exists a uniform cover time vector $\overline{cv}$ for which there is an execution of* MZP2 *that does not converge in strictly less than $2 \cdot cv - 2$ events.*

Proof. We consider an integer g such that $g \cdot (n-3) \geq decay + 1$, and we define specific sequences as follows :

- $U = (3\ BS) \dots (n-1\ BS)$.
  Agents 3 to $n-1$ meet the base station once.
- $V = [(2\ 3) \dots (2\ n-1)] \cdot [(3\ 4) \dots (3\ n-1)] \cdot \dots \cdot (n-2\ n-1)$
  All mobile agents, except agent 1, meet each other once.
- $W = (1\ 3) \dots (1\ n-1)$.
  Agent 1 meets every other mobile agent, except agent 2, exactly once.
- $X = U \cdot V^g \cdot W \cdot (2\ BS)(1\ 2)(1\ BS)$

We build a schedule $\mathcal{S}$ by repeating X infinitely many times : $\mathcal{S} = X^\omega$. We choose the same cover time, $cv = |X|$, for all agents. A simple computation shows that $cv = 2n - 3 + g \cdot \frac{(n-3)(n-2)}{2}$. By construction, $\mathcal{S}$ satisfies the cover time property.

Now we prove that the execution of MZP2, induced by the sequence $\mathcal{S}$, does not converge before the first $2 \cdot cv - 2$ events. At the end of the first U in $\mathcal{S}$, agents 3 to $n-1$ have successively met BS and transferred their values. Thus, all variables $accumulation_x$, for $3 \leq x \leq n-1$, equal 1. In the sequence V^g, each agent $x \neq 1$ is involved in $g \cdot (n-3) \geq decay + 1$ meetings. Hence, thanks to the decay mechanism, at the end of the first V^g, every agent x, from 2 to $n-1$, has its variable $accumulation_x$ reset to 0. As a consequence, there is no transfer from agent 1 to any other mobile agent during the sequence W that follows V^g. Then, during the sequence $(2\ BS)(1\ 2)(1\ BS)$, agent 2 receives the input value v of 1. From this point, agent 2 cannot transfer v to any other agent but BS. This event can happen cv events later (event $(2\ BS)$ in the second X of $\mathcal{S}$). Therefore, the value v is delivered to BS exactly after the $(2 \cdot cv - 2)$-th event of the schedule. □

4.5 Bounded Memory

Up to now, we have assumed that mobile agents have an "unbounded" ($O(n)$) memory. In this section, we discuss the case of bounded (constant) memory, i.e., a memory size independent of the number of agents. We assume that the memory of an agent can hold at most k data values, with $k \geq 1$. Both MZP1 and MZP2 can be adapted to this assumption. Any transfer of values between mobile agents is now limited by the available memory and the transfer may be *partial*. During an event, as much as possible values are transferred. Note that all the data values are equivalent for the data collection problem, thus it is unnecessary to precise which values are actually transferred. In the adapted version of MZP1, once an agent has transferred some values, even partially, it becomes inactive and cannot receive any other value. For every agent x, the values held by x are stored in a dynamic array $values_x$, whose size is denoted by $size(values_x)$. By definition, we have $size(values_x) \leq k$. Algorithm 4 presents an adaptation of MZP1, but the same idea can be applied to MZP2. For the sake of clarity, we do not present in the code the management of the dynamic array $values_x$. We denote by MZP1-BM (resp. MZP2-BM) the bounded-memory version of MZP1 (resp. MZP2).

Algorithm 4: Modified ZebraNet Protocol 1 - Bounded memory

```
when x meets BS do
    <x transfers its values to BS>
    accumulation_x := accumulation_x + 1
    distance_x := 0
end when
when x meets y ≠ BS do
    count := min(size(values_x), k − size(values_y))
    if accumulation_x < accumulation_y ∧ active_y ∧ count > 0 then
        <x transfers count values to y>
        active_x := false
    end if
    distance_x := distance_x + 1
    if distance_x > decay then
        if accumulation_x ≠ 0 then
            accumulation_x := accumulation_x − 1
        end if
        distance_x := 0
    end if
end when
```

For both MZP1-BM and MZP2-BM, it appears that the proofs given in the previous sections (Secs. 4.3, and 4.4) are still applicable. Indeed, the memory size tightens the constraints on transfers, but do not fundamentally affect the structures of the executions of the algorithms MZP1 and MZP2. Still, we sketch the proofs for MZP1-BM and MZP2-BM.

Proposition 11 (Bounds to the MZP1-BM complexity)**.** *For any graph G of $n \geq 1$ agents, for any cover time vector $\overline{cv}$, for any decay ≥ 1, any execution of* MZP1-BM *converges in no more than $\sum_{x \in G^*} cv_x - 2 \cdot (n-2)$ events.*

For any graph G of $n \geq 4$ agents, for any decay ≥ 1, there exists a uniform cover time vector $\overline{cv}$ for which there is an execution of MZP1-BM *that converges in $(n-1) \cdot cv - 2 \cdot (n-2)$ events.*

Proof. The fact that MZP1-BM converges is due to the fact that the set of active agents cannot increase. As in MZP1, once the set of active agents remains constant, there cannot be any transfer between any two mobile agents. Since all mobile agents meet BS in the next cv_{max} events, the protocol converges.

The upper bound to the complexity of MZP1-BM is computed by observing the path followed by the last delivered value v, i.e., the sequence of the mobile agents that successively carry v. The memory size does not affect the fact that a mobile agent in this path cannot appear twice, thanks to the *active* bit, nor the fact that a mobile agent x in this path holds v for at most $cv_x - 1$ or $cv_x - 2$ consecutive events. The same construction as in the proof in Sec. 4.3 shows that any execution of MZP1-BM converges in no more than $\sum_{x \in G^*} cv_x - 2 \cdot (n-2)$ events.

The lower bound to the complexity of MZP1-BM is obtained thanks to the same schedule as in Sec. 4.3. Indeed, applying this schedule to an initial configuration gives an execution in which each agent holds at most one value, which is compatible with the assumption $k \geq 1$. □

Proposition 12 (Bounds to the MZP2-BM complexity)**.** *For any graph G of $n \geq 1$ agents, for any cover time vector $\overline{cv}$, for any decay ≥ 1, any execution of* MZP2-BM *converges in no more than $2 \cdot cv_{max} - 2$ events.*

For any graph G of $n \geq 4$ agents, for any decay ≥ 1, there exists a uniform cover time vector $\overline{cv}$ for which there is an execution of MZP2-BM *that does not converge in strictly less than resp. $2 \cdot cv - 2$ events.*

Proof. During the first cv_x events, agent x either transfers its input value v to BS or to another mobile agent y. In the second case, the transfer occurs in the first $cv_x - 1$ events. At this point, agent y is inactive and cannot transfer v to any other agent, but BS, which is done in the next $cv_y - 1$ events. Thus MZP2-BM also converges in no more than $2 \cdot cv_{max} - 2$ events.

The lower bound to MZP2-BM is obtained thanks to the same schedule as in Sec. 4.4. Indeed, applying this schedule to an initial configuration gives an execution in which each agent holds at most one value, which is compatible with the assumption $k \geq 1$. □

4.6 Remarks

(MZP1 and MZP2 vs. a "trivial" protocol). One can notice that the worst case complexities of MZP1 and MZP2 are worse than for a very trivial protocol, in which each agent can transfer its value only directly to the base station (its complexity is cv_{max}). By the same measure, this protocol is better than the protocol that is practically used by ZebraNet (the one which does not converge according to Sec. 4.2). The reason is that the worst case value is obtained when all agents have the same cover time, cv_{max} (see Lem. 8 and 10). In practice, and in particular concerning the ZebraNet project, there are most likely different types of agents and the effective complexity, as we demonstrate

below, is most likely better for MZP1, MZP2 (and even for the original protocol) than for the trivial one. In the actual settings of ZebraNet, there are some frequent periods of time when the base station may be inactive and some zebras may be sleeping or ill and thus motionless. In contrast with the trivial protocol, during such periods, the other protocols perform better.

Let us give a more quantitative example. Consider a population of zebras, distributed more or less into two major categories : (1) the healthy zebras which are very mobile, very often near the base station and near the other zebras; and (2) the ill zebras which are most of the time motionless and away from the base station. Thus, assume that there are h ill zebras and each has a very high (near infinite) cover time, at least cv_h. There are s healthy zebras and each has a much smaller cover time, at most cv_s. Thus, $cv_s \ll cv_h$ and $s + h = n$. In addition, for the sake of simplicity, assume that $cv_s < decay \ll cv_h$ and that for every healthy zebra x, $size(values_x) \geq h$, i.e., every healthy zebra has enough memory to collect the values of all the ill zebras.

Now, compare the expected complexities of the trivial protocol with the protocols MZP1 and MZP2. The trivial protocol has a very large complexity of $cv_{max} \geq cv_h$, since the only transfers are towards the base station. The protocol MZP1, intuitively, will have an average complexity of the order of less than $s \cdot cv_s$. This is an upper bound on the number of events of some prevalent scenario, where a value of an ill zebra is relayed to a healthy active zebra, then, every cv_s events, to another healthy active zebra, and finally to the base station. If there are much more healthy mobile zebras than ill and motionless ones, finding an active healthy zebra is not a problem. The scenario in which a value of an ill zebra is relayed by ill zebras (this is the type of scenario that gives the near worst case complexity), is not likely to happen when the mobility of zebras is modeled by random walks. In consequence, the weight of such scenarios will be very small in the computation of the average complexity.

On the same population (distributed into two categories), MZP2, intuitively, will have an average complexity less than $3 \cdot cv_s$. This is a worst case complexity of a prevalent scenario of MZP2 where, first, the healthy zebras meet the base station (in at most cv_s events) and augment their accumulation variables. Then, in at most cv_s events, a value of an ill zebra is relayed to a healthy zebra and then, in at most additional cv_s events, to the base station.

(MZP1 vs. MZP2). In spite of the worst case complexity results, MZP1 has some advantages over MZP2. First, note that MZP1 is a multi-hop protocol, in contrast with MZP2 which is a two-hop one. Hence, MZP1 approximates better the original ZebraNet protocol (which is also multi-hop) than MZP2. Second, note that, although the example above describes a population where MZP2 performs better than MZP1, there is a large class of populations where, at the contrary, MZP1 exhibits a better performance than MZP2. Consider, for instance, a population which is distributed more or less into several (more than two) major categories. For example, there are young healthy, old healthy, young ill and old ill zebras' categories. Then, consider scenarios where, at the beginning, a value of an old ill zebra is transferred to some young ill zebra. Then, with MZP2, this young ill zebra is the only zebra that could deliver the value to the base station. However, with MZP1, the value is likely to

be transferred to an active young and healthy zebra. Thus, it may be delivered to the base station faster than with MZP2. It could be interesting to make a deeper investigation on the comparison between MZP1 and MZP2.

Chapter 5

Consensus

5.1 Introduction

Consensus is a decision problem, classical in fault-tolerant distributed computing. In this problem, each process (agent in our case) is given initially a value and has to take eventually an irreversible decision (*termination* condition). Processes must decide on a common value depending on the input values, according to *agreement* and *validity (non-triviality)* conditions [74, 62, 11].

Consensus-related problems are relevant to mobile sensor networks in many different contexts like for example, flocking (see, e.g., [35]), swarm formation control (see, e.g., [75]), distributed sensor fusion (see, e.g., [70]) and attitude alignment (see, e.g., [61]). See also [72, 69, 71] for surveys and references on consensus-related problems in mobile wireless networks.

A fundamental result by Fisher, Lynch and Paterson [47] states that in the classical asynchronous message passing model, no deterministic algorithm for consensus exists, even in the case of a unique possible crash (halting) failure. It is not surprising that the same impossibility holds in the model of population protocols. This model is fundamentally asynchronous, which is one of the main reasons for the result in [47]. However, some inherent characteristics of population protocols make consensus even more difficult. The agents are uniform (indistinguishable and executing the same code). They have a constant memory size and thus cannot neither obtain nor store labels or any other information depending on the network size. Agents communicate by asynchronous interactions such that each interaction is between a couple of agents. No broadcast communication is available. Due to all these limitations, the agents are unable to detect which other agents are present but not interacting, even if no crash failure is possible. Hence, in population protocols, even with the assumption of absence of failures, consensus is impossible (Sec. 5.2).

Like in the message passing model, it seems interesting to study what is missing for solving consensus in population protocols. We adopt the point of view of Chandra and Toueg [32] for defining the possible missing information under the form of oracles, i.e., specific behaviours. Recall that, an oracle can be thought as a collection of modules able to provide each process with some information, hopefully useful to solve a given problem. The failure detectors [32] usually provide each process with failure-related information. Specifically, the failure detectors of [32] cannot be used in our case, because they furnish

lists of process identifiers (estimated to have crashed). As already mentioned, identifiers are absent in population protocols (due to the constant size memory requirement).

Nevertheless, several identity-free oracles exist in the literature. The failure detector introduced in [37] outputs a boolean value at every processes, and solves the $(n-1)$-set agreement problem in n-process message passing system. This failure detector has been shown to be the weakest to solve this problem, which is weaker than consensus. This involves that such an oracle cannot be helpful to solve consensus in population protocols. Another type of oracles proposed in [68, 67] (and used, e.g., in [21, 22]) to deal with anonymity, provide information on the number of crashed processes (bounded by $f < n$), and, for the same reason of constant agent state space, cannot be used in the framework of population protocols. A so called "heartbeat" failure detector proposed in [3] requires to maintain unbounded counters at every process, and thus, again, is not suitable in our case. Some other failure detectors that are used to solve consensus and adapted to anonymous systems, like $A\Omega$ [22], $A\mathcal{L}$ and $A\Sigma'$ [24, 23], but these provide information whose size depends on the number of agents. In addition, in message-passing system, these oracles are used in combination with other powerful model assumptions and capabilities (as possible "terminating" broadcast and unbounded process memory) which are unavailable in our case.

This is the reason why we introduce a new class of oracles. The constraints we had in mind, when designing these oracles, are basically to make them implementable with minimum external assumptions on the system. In short, these oracles provide information related only to the past schedules. To provide this information, an oracle outputs a boolean value at every agent (similarly to the failure detector in [37]). The oracle is not required to provide this information at the precise time when it appears, but only eventually, at least once and at some agent (i.e., it is unreliable in this sense). Finally, the proposed oracles are anonymous, in the sense that a permutation of the agents identities does not affect the possible output of these oracles (see Sec. 5.3 for precise definitions).

The Problem

Consider a population protocol $\mathcal{A}$ with initial values $\mathcal{V}$. We assume that the agents have an instruction *decide* which causes them to decide irreversibly on some value in $\mathcal{V}$.

The population protocol $\mathcal{A}$ (possibly using an oracle) is said to solve the *consensus* problem if, for each complete graph G, for each initial configuration γ, for any legal execution $H[\gamma]$, it satisfies the following: *i. (termination)* every agent eventually decides in the execution $H[\gamma]$; *ii. (agreement)* two agents cannot decide on different values; *iii. (validity)* if all the agents have the same initial value v, then an agent can only decide on v.

We now define the *symmetric consensus* problem that will be studied here. The protocol $\mathcal{A}$ is said to solve the symmetric consensus problem if it solves the consensus problem and, in addition, for each complete graph G, it satisfies the following additional condition: *iv. (symmetry)* for any legal execution $H[\gamma]$, for any permutation $\alpha \in \mathfrak{S}G$ of the vertices, any agent decides on the same value in the execution $H[\gamma]$ and in $H[\gamma\alpha]$.

Intuitively, the decision value in an execution does not depend on the distribution of the initial values between the agents.

Context

In this chapter, we study the symmetric consensus in the following context. We focus on the family of complete graphs. The agents are initialized with input values, on which they have to decide. The fair executions are the ones whose underlying schedules satisfy the weak fairness condition. Recall that a schedule S is weakly fair when every agent meets indirectly with every other agents infinitely often during S. In other words, for each (ordered) pair of agents (x, y), S contains infinitely many segments u during which y meets indirectly with x. Refer to Chap. 3, Sec. 3.2 for further details.

Overview

In Sec. 5.2, it is shown that there is no population protocol that solves the consensus problem (hence, including the symmetric consensus) in the current context. This impossibility suggests the introduction of oracles (i.e. behaviours) to study the hardness of the symmetric consensus. In Sec. 5.3, the class $Mnemosyne$ of oracles is defined. Intuitively, an oracle of $Mnemosyne$ notifies the agents whenever it finds some predefined schedule patterns in their causal pasts. A specific oracle in $Mnemosyne$, namely $DejaVu$, is introduced: this oracle notifies an agent whenever this agent has indirectly met with every other agent (at least once). In Sec. 5.4, it is shown that $DejaVu$ is sufficient to solve the symmetric consensus problem. Then, in Sec. 5.5, it is shown that this oracle is "necessary" to solve the symmetric consensus problem, in the sense that, it is the weakest oracle in $Mnemosyne$ able to solve this problem. The definition of $Mnemosyne$ may look intricate at first sight. Yet, in Sec. 5.6, it is shown that the class $Mnemosyne$ derives from axioms which are very natural in the context of population protocols.

5.2 Impossibility of Consensus without Oracle

We first show that the consensus problem is impossible without an oracle. This obviously shows that the symmetric consensus problem is also impossible. The proof relies on the well-known partitioning argument.

Proposition 13. *Under weak fairness, there is no population protocol that solves the consensus problem over the complete graphs.*

Proof. Assume that there exists a population protocol $\mathcal{A}$ that solves the consensus problem over all complete graphs. Pick a complete graph G of $2{\cdot}n$ agents (vertices), and select two complete subgraphs G_0, G_1 of n agents each. Let γ be the initial configuration of $\mathcal{A}$ corresponding to the agents in G_0 (resp. G_1) having the initial consensus value 0 (resp. 1). Let S_v be a weakly fair schedule (crash free) over G_v. By the validity condition of the consensus problem, in the execution $S_v[\gamma]$, all agents in G_v decide on the value v. Let S'_v be a finite prefix of S_v such that all the agents in G_v decide (on v) in the finite execution $S'_v[\gamma]$. Let S'' be any weakly fair (crash-free) extension of the concatenated

schedule $S'_0 \cdot S'_1$. Then, in the execution $S''[\gamma]$, the agents in G_0 decide on 0, and the agents in G_1 decide on 1; whence a contradiction with the agreement condition. □

Note that this results is easily extended for any non-simple family of graphs. A family $\mathcal{F}$ is non-simple if there exist graphs $G_1, G_2, G \in \mathcal{F}$ such that G_1 and G_2 are disjoint subgraphs of G.

5.3 Class of Oracles

Prop. 13 motivates the use of oracles. In this section, we define a specific class of oracles ($Mnemosyne$), in which we will look for the weakest oracle able to solve the symmetric consensus. The $Mnemosyne$ class lives in a larger class $\mathcal{U}$ of oracles that we first present.

Anonymous Binary Oracles

Roughly speaking, the oracles in $\mathcal{U}$ observe the schedule S of events, and give to each agent some information, in $\{0, 1\}$, about S. This is similar to the situation in [32], where a failure detector gives information (list of identifiers) about the failure pattern.

Formally, for every oracle O in the class $\mathcal{U}$, its domain is the complete graphs, its input alphabet is a singleton (i.e. no input), and its output alphabet is $\{0, 1\}$. Note that since the input alphabet is a singleton, an input history $H = (S, h)$ reduces to its underlying schedule S.

In addition, the oracles of $\mathcal{U}$ are anonymous. Roughly speaking, an oracle can be seen as a collection of local modules, or blackboxes, each of them being attached to an agent in the population. But, saying that an oracle is anonymous means that there are no preferred ways of mapping these modules to the agents.

Formally, an oracle O is *anonymous* if, for every graph G, every fair schedule S, the set of legal output histories can be written

$$O(G, S) = \bigcup_{\sigma \in \mathfrak{S}G} O(G, \sigma, S) \tag{5.1}$$

where $\mathfrak{S}G$ denotes the group of permutation of the vertices of G (i.e. the automorphism group of the graph G since G is complete), and the $O(G, \sigma, S)$'s are sets of histories satisfying

$$\forall \alpha \in \mathfrak{S}G,\ H \in O(G, \sigma, S) \Leftrightarrow \alpha H \in O(G, \alpha\sigma, \alpha S) \tag{5.2}$$

Recall that αS denotes the schedule obtained by replacing each event e with the event $\alpha(e)$, and αH denotes the history obtained from H by transporting the values likewise.

Intuitively, the set $O(G, \sigma, S)$ represents the legal histories when the local blackboxes are mapped to the agents according to σ. The condition in Eq. 5.2 implies that if a history H is legal for the mapping σ, then any of its permutation αH is also legal for the corresponding mapping $\alpha\sigma$. In other words, the sets $O(G, \alpha, S)$ and $O(G, \beta, S)$ are the same modulo a permutation $\sigma = \alpha\beta^{-1}$ of the vertices.

An example of non-anonymous oracle would be the oracle that outputs 1 at a unique and specified agent λ, and zero everywhere else. The anonymous counterpart of this oracle would be the oracle that outputs 1 at a unique agent drawn arbitrarily from the population.

Mnemosyne

We now define a subclass $Mnemosyne \subseteq \mathcal{U}$ of oracles. Intuitively, an oracle of $Mnemosyne$ outputs 1 at some agent x if the oracle recognizes a predefined pattern in the causal past of x.

Each oracle O in $Mnemosyne$ is defined by a family of (possibly empty) sets $Cones(O, G, \sigma, x)$ of finite schedules for every complete graph G, every permutation σ of the vertices and every agent x in G. The set $Cones(O, G, \sigma, x)$ represents the patterns that will be looked for in the causal past of x. In addition, it satisfies the following properties:

- *i. (anonymous)* for every permutation $\alpha \in \mathfrak{S}G$ of the agents, $K \in Cones(O, G, \sigma, x)$ if and only if $\alpha K \in Cones(O, G, \alpha\sigma, \alpha(x))$
- *ii. (cone)* every schedule K in $Cones(O, G, \sigma, x)$ is a past cone at x.
- *iii. (saturation)* if $K \in Cones(O, G, \sigma, x)$ and $K \simeq K'$, then $K' \in Cones(O, G, \sigma, x)$
- *iv. (unavoidable)* for every (weakly) fair schedule S, there exists an agent x such that S contains, as a commuting factor, a schedule from $Cones(O, G, \sigma, x)$.

A history H belongs to $O(G, \sigma, S)$ if and only if it satisfies

- *i. (safety)* If H outputs 1 at x in some event p in S, then the prefix $S \uparrow p$ contains, as a commuting factor, some schedule from $Cones(O, G, \sigma, x)$.
- *ii. (liveness)* the history H eventually outputs 1 at some agent in some event during S.

Intuitively, the property *(safety)* ensures that if O outputs 1 at x, then the corresponding prefix actually contains (as a commuting factor) a schedule from $Cones(O, G, \sigma, x)$. The property *(liveness)* ensures that at least one agent is eventually notified about this fact. Note that, thanks to the condition *(unavoidable)*, it is always possible for an oracle to output 1 at some agent in any schedule. The property *(saturation)* implies that an oracle in $Mnemosyne$ is not able to distinguish schedules like $S_1 = e_1 e_2$ and $S_2 = e_2 e_1$, where e_1 and e_2 are independent events. The schedule S_1 (resp. S_2) means that *in real-time* the event e_1 (resp. e_2) occurs before e_2 (resp. e_1). Hence, the property *(saturation)* expresses the fact that the oracle has no access to a real-time clock. The condition *(anonymous)* on the $Cones$ sets is required for the oracle to satisfy the anonymity condition of $\mathcal{U}$ (Eq. 5.2). Note that the set $Cones(O, G, \sigma, x)$ is possibly empty, which means that it is possible, a priori, for the oracle to permanently output 0 at x.

$DejaVu$ oracle

The oracle $DejaVu$ in the $Mnemosyne$ class is defined as follows. A schedule K belongs to $Cones(DejaVu, G, \sigma, x)$ if and only if K is a past cone at x such that $supp(K) = G$. Intuitively, thanks to the properties of the $Mnemosyne$ class, the $DejaVu$ oracle can output 1 at an agent x when x has indirectly met all the agents.

5.4 Symmetric Consensus with $DejaVu$

The purpose of this section is to show that $DejaVu$ is strong enough to solve the symmetric consensus. A simple protocol using $DejaVu$ is presented under the form of pseudo-code (Alg. 5), which is equivalent to the representation using transition rules.

We denote by $\mathcal{V}$ the set of initial consensus values. Every agent x has the following variables: an estimate of the consensus value val_x (initially set to a value in $\mathcal{V}$), a boolean flag $decided_x$ (initially $false$), and a read-only boolean variable $done_x^{DV}$ which is output by the oracle $DejaVu$. We assume that the set of consensus values is totally ordered. When two agents x and y meet, they both select the minimum of val_x and val_y as a new estimate of the consensus value. An agent x decides on its estimate when either the oracle $DejaVu$ outputs true, or it meets with an agent that has already decided; the agent then sets its flag $decided_x$ to $true$.

Algorithm 5: Symmetric consensus with $DejaVu$

```
1 done_x^DV : output of the oracle DejaVu at x;
2 Initialization:;
3     val_x ← a value in V;
4     decided_x ← false;
5 On a meeting event (x, y) of the agents x and y:;
6     val_x ← min(val_x, val_y);
7     if ¬decided_x ∧ (done_x^DV ∨ decided_y) then
8         decide on val_x;
9         decided_x ← true;
```

Lemma 1 (Termination and Validity)**.** *Let $H \in DejaVu(G, \sigma, S)$ be a legal history and γ be an initial configuration. Then, in the execution $H[\gamma]$, every agent eventually decides on some initial value present in γ.*

Proof. Since an agent x can only decide on its estimate val_x, and since every update of val_x assigns a value of some agent, x can only decide on a value present in γ. The liveness property of the oracle $DejaVu$ implies that the oracle eventually outputs $true$ at some agent x, which thus decides. Thanks to weak fairness, every agent will eventually indirectly meet with x, and decides too (if it has not decided already). □

Lemma 2. *Let $H = (S, h)$ be any history with values in $\{0, 1\}$ on the complete graph G, and γ be an initial configuration. Consider the prefix $S \uparrow p$ of S for some event p in S, and let x be an agent involved in p. Then, at the end of the*

finite execution $H|_{S\uparrow p}[\gamma]$, the value of val_x at x is equal to the minimum of the initial values of the agents in the support of the causal past of p.

Proof. For any occurrence of a event p in S, for any agent z involved in p, we denote by $val(p, z)$ the value of val_z right after p. We denote by $val(\epsilon, z)$ the initial value of the agent z.

Let x, y be the agents involved in the event p. Let p_x (resp. p_y) be the immediate predecessor[1] of p in $Past(p)$ that involves the agent x (resp. the agent y). If such an immediate predecessor does not exist (i.e. p is the first event involving x (resp. y)), then we set $p_x = \epsilon$ (resp. $p_y = \epsilon$). By line 6 in Alg. 5, $val(p, x) = \min(val(p_x, x), val(p_y, y))$. By iterating, we get $val(p, x) = \min\{v_z, z \in supp(Past(p))\}$. □

Lemma 3. *Consider Alg. 5 using $DejaVu$. Let $H \in DejaVu(G, \sigma, S)$ be a legal history of $DejaVu$, γ be an initial configuration. In the execution $H[\gamma]$ of Alg. 5, if some agent x' decides in some event p', then $supp(Past(p')) = G$.*

Proof. When x' decides, it is either because of the meeting with an agent which has already decided, or because the oracle has output 1 at x'_i (Alg. 5, line 7). Hence, there is an event p_i (in S) involving some agent x such that $p \rightsquigarrow p'$ and the oracle has output 1 at x during p (note that p and p' may be the same event).

By the safety property of $DejaVu$, $S \uparrow p$ contains, as a commuting factor, some schedule from $Cones(DejaVu, G, \sigma, x)$. Hence, $supp(Past(p')) = supp(Past(p)) = G$, by the definition of $DejaVu$. □

Proposition 14. *If Alg. 5 uses $DejaVu$, then it solves the symmetric consensus.*

Proof. The termination and validity conditions are satisfied thanks to Lem. 1. The agreement and symmetry conditions are satisfied thanks to Lem. 2 and 3. □

5.5 Weakest Oracle for Symmetric Consensus

In this section, we prove that any oracle O in $Mnemosyne$ allowing to solve symmetric consensus can be used to implement $DejaVu$. Thus, together with the result of Sec. 5.4, this proves that $DejaVu$ is the weakest oracle in $Mnemosyne$ to solve symmetric consensus.

Lemma 4. *Let $\mathcal{A}$ be a population protocol that solves the consensus problem using an oracle O in $Mnemosyne$. For every graph G, and every permutation $\sigma \in \mathfrak{S}G$, there exists an agent x such that $Cones(O, G, \sigma, x) \neq \emptyset$.*

Proof. On the contrary, if for some graph G, and some σ, for every agent x, $Cones(O, G, \sigma, x) = \emptyset$. Then, there is a contradiction with the property *(unavoidable)*. □

[1] immediate means that, if p' involves x and $p_x \rightsquigarrow p' \rightsquigarrow p$, then $p' = p_x$ or $p' = p$.

The following crucial lemma shows that the sets $Cones$ defining O are subsets of those defining $DejaVu$ respectively. Note that, it is still possible for some $Cones(O, G, \sigma, x)$ to be empty. The lemma states that if it is not, then the support of each of its schedules is the entire graph G.

Lemma 5. *Let $\mathcal{A}$ be a population protocol that solves the symmetric consensus problem over all complete graphs using an oracle O in Mnemosyne. Then, for every complete graph G, every permutation σ, and every agent x in G, $Cones(O, G, \sigma, x) \subseteq Cones(DejaVu, G, \sigma, x)$.*

Proof. In this proof, for sake of clarity, we use the same notation for the initial value of an agent, and the corresponding initial state. Assume that there is some schedule $K \in Cones(O, G, \sigma, x)$ that is not in $Cones(DejaVu, G, \sigma, x)$, i.e., K is a past cone at x whose support $D = supp(K)$ is a strict subgraph of G.

By Lemma 4, for some agent y, the set $Cones(O, D, \sigma, y) \neq \emptyset$. Let $\alpha \in \mathfrak{S}G$ that swaps x and y and $\beta = \alpha\sigma$. Then, by the anonymity property of the cones set, $Cones(O, D, \beta, x) \neq \emptyset$. Thus, pick some $K' \in Cones(O, D, \beta, x)$.

Let $S = K \cdot K' \cdot S'$ be any weakly fair extension of $K \cdot K'$ on D. We build a history H with schedule S as follows: the history always outputs 0 everywhere except at x, for which it permanently outputs 1 only after $K \cdot K'$. Since $K' \in Cones(O, D, \beta, x)$, we have $H \in O(D, \beta, S)$, i.e. H is a legal history of O on D.

For any initial configuration γ on D, we have an execution $H[\gamma]$ of $\mathcal{A}$ in which every agent in D decides. By the validity property of the consensus, if all the agents have the same initial value 0 (resp. 1), then all agents decide on 0 (resp. 1).

Hence, there exist two initial configurations γ_0 and γ_1 on D such that, for some agent a in D, $\gamma_0(a) = 0$, $\gamma_1(a) = 1$ and for every $z \in D - \{a\}$, $\gamma_0(z) = \gamma_1(z)$, and the agents decide on the value 0 (resp. 1) in the execution $H[\gamma_0]$ (resp. $H[\gamma_1]$).

In particular, x decides on 0 in $H[\gamma_0]$ after some event p_0 in S, and decides on 1 in $H[\gamma_1]$ after some event p_1 in S. Let L be the a prefix of S that has both $S \uparrow p_0$ and $S \uparrow p_1$ as prefixes. By the end of the finite executions $H|_L[\gamma_0]$, $H|_L[\gamma_1]$, x decides on the values 0 and 1 respectively.

We can extend $H|_L$ to get a weakly fair legal history H' of O on the graph G as follows. Consider the schedule $L \cdot S''$ for some weakly fair schedule S'' on G. In L, the history H' outputs the same values as $H|_L$; and in S'', it outputs 0 everywhere except at x, for which it outputs 1. Since $K \in Cones(O, G, \sigma, x)$, we have $H' \in O(G, \sigma, x)$, i.e., H' is a legal history of O on G.

For $v \in \{0, 1\}$, let g_v be the initial configuration on G such that g_v is equal to γ_v on D, and 1 elsewhere. In $H'[g_0]$, agent x decides by the end of L. The support of the causal past of the event preceding its decision, is included in D. Hence, since g_0 and γ_0 are equal on D, x decides 0 in $H'[g_0]$. For similar reasons, x decides 1 in $H'[g_1]$. Now pick an agent y in $G - D$, and let g be the initial configuration obtained from g_0 by permuting the values of a and y. In other words, $g(a) = g_0(y) = 1$, $g(y) = g_0(a) = \gamma_0(a) = 0$, and, for every $b \in G - \{a, y\}$, $g(b) = g_0(b)$. The restriction of g to D is equal to γ_1. Hence, in $H'[g]$, the agent x decides on the value 1. On the other hand, since the

protocol solves the symmetric consensus, x decides on the value 0; whence a contradiction. □

Theorem 1 (Weakest Oracle)**.** *The DejaVu oracle is the weakest oracle in the Mnemosyne class that can be used to solve symmetric consensus.*

$$\forall O \in Mnemosyne,\ O \text{ solves symmetric consensus } \Rightarrow DejaVu \preccurlyeq O \quad (5.3)$$

Proof. Consider such an oracle O. By Lemma 5, we have $Cones(O, G, \sigma, x) \subseteq Cones(DejaVu, G, \sigma, x)$ for every triple (G, σ, x). We claim that O is a sub-behaviour of $DejaVu$.

Indeed, let $H \in O(G, \sigma, S)$. By the liveness property of O, H eventually outputs 1 in some event; hence H satisfies the liveness property of $DejaVu$. On the other hand, if H outputs 1 at x in some event p, then, by the safety property of O, the prefix $S \uparrow p$ contains, as a commuting factor, some schedule $K \in Cones(O, G, \sigma, x)$. Since $K \in Cones(DejaVu, G, \sigma, x)$, H also satisfies the safety property of $DejaVu$. Hence, $H \in DejaVu(G, \sigma, S)$. In other words, $O(G, S) \subseteq DejaVu(G, S)$, i.e., O is a sub-behaviour of $DejaVu$. In particular, $DejaVu \preccurlyeq O$ (see Chap. 3, Sec. 3.5, Prop. 4). □

5.6 Derivation of Mnemosyne

In this section, we show that *Mnemosyne*, although quite intricate at first sight, derives from a limited number of natural axioms.

Axioms

We define a subclass $\mathcal{V} \subseteq \mathcal{U}$ of anonymous binary oracles. Every oracle O in $\mathcal{V}$ is defined over complete graphs, have no input values[2], has output values $Out(O) = \{0, 1\}$, and satisfy the following conditions (see comments below):

- *(anonymous)* There is a family $\{O(G, \sigma, S)\}$ of history sets such that

$$O(G, S) = \bigcup_{\sigma \in \mathfrak{S}G} O(G, \sigma, S)$$
$$\forall \alpha \in \mathfrak{S}G,\ H \in O(G, \sigma, S) \Leftrightarrow \alpha H \in O(G, \alpha\sigma, \alpha S)$$

- *(no future)* For every $H \in O(G, \sigma, S)$, for every prefix $L \sqsubset S$, for every extension $S' \sqsupset L$, there exists a history $H' \in O(G, \sigma, S)$ such that $H|_L = H'|_L$.

- *(unreliable delay) i.* for every $H \in O(G, \sigma, S)$, for every finite schedule L, let H' be the history with schedule L that outputs 0 during L and the same values as H during S. Then, $H' \in O(G, \sigma, L \cdot S)$.

  ii. Also, for every $H \in O(G, \sigma, S)$, for every agent x, for every (occurrence of) event p_0 in S involving x, let $p_0, p_1, \ldots$ be the successive events involving the agent x, and $v_0, v_1, \ldots$ the corresponding output of the history H at x. Then the history H' which outputs the same values as H except at $p_0, p_1, p_2, \ldots$ where it respectively outputs $0, v_0, v_1, \ldots$, belongs to $O(G, \sigma, S)$.

[2] i.e. $In(O)$ is reduced to a singleton.

- *(causality)* for every $H \in O(G, \sigma, S)$, for any permutation $\tau \in \mathfrak{S}\mathbb{N}$ such that $S\tau \simeq S$, $H\tau \in O(G, \sigma, S\tau)$.
- *(liveness)* Every history in $O(G, \sigma, S)$ eventually outputs 1 at some agent in some event occurring in S.
- *(schedule dependent)* For any $H_1, H_2 \in O(G, \sigma, S)$, for any event e in S involving the agent x, let v_1 the value output by H_1 at x in e, then the history H which outputs the same values as H_2 everywhere except at x in e where it outputs v_1, belongs to $O(G, \sigma, S)$.

The condition *(anonymous)* is exactly the condition characterizing the oracles of $\mathcal{U}$ (Sec. 5.3); thus $\mathcal{V}$ is a subclass of $\mathcal{U}$. The condition *(no future)* states the oracle cannot foresee the future events of the schedule. The condition *(unreliable delay)* states that the signal of the oracle can be delayed arbitrarily. The condition *(causality)* states the oracle has no access to a real-time clock, i.e., it cannot distinguish independent events. The condition *(liveness)* simply states that the oracle eventually indicates something. Finally, the condition *(schedule dependent)* states that the value output by the history in some event only depends on the past schedule, and not on the previous output values.

Derivation

We now show how to derive *Mnemosyne* from the previous axioms. The idea is to extract from the histories of an oracle O a family of schedule sets $\{Z_0(O, G, \sigma, x)\}$ that will eventually play the role of the sets $Cones(\dots)$. First, we define the following sets.

Definition 3 (Sets $Z(O, G, \sigma, x)$)**.** *Given an oracle O in $\mathcal{V}$, a finite schedule L belongs to $Z(O, G, \sigma, x)$ if and only if $x \in supp(L)$ and there exist an extension $S \sqsupset L$ and a history $H \in O(G, \sigma, S)$ that outputs 1 at x in the last event in L involving x.*

Intuitively, the oracle gives information about the past when it raises its signal from 0 to 1. Hence, these schedule sets somehow represent the patterns that the oracle observes. Note that, these are not yet candidates for being the family *Cones*. The following lemma highlights the properties of the family Z.

Lemma 6. *For any oracle O in $\mathcal{V}$, the family $\{Z(O, G, \sigma, x)\}$ satisfies the following properties:*

- (anonymous) *for any permutation $\alpha \in \mathfrak{S}G$, $L \in Z(O, G, \sigma, x)$ if and only if $\alpha L \in Z(O, G, \alpha\sigma, \alpha(x))$.*
- (saturation) *if $L \in Z(O, G, \sigma, x)$ and $L \simeq L'$ then $L' \in Z(O, G, \sigma, x)$.*
- (extension) *if $L \in Z(O, G, \sigma, x)$, then for any finite schedules A and B, $A \cdot L \cdot B \in Z(O, G, \sigma, x)$.*

Proof. (anonymous). We prove that if $L \in Z(O, G, \sigma, x)$ then for any permutation α, $\alpha L \in Z(O, G, \alpha\sigma, \alpha(x))$. Indeed, by definition, there exists an extension $S \sqsupset L$ and a history $H \in O(G, \sigma, S)$ which outputs 1 at x in the last event in L involving x. Since $O \in \mathcal{V}$ is anonymous, we have $\alpha H \in O(G, \alpha\sigma, \alpha S)$. But αH outputs 1 at $\alpha(x)$ in the last event in αL involving $\alpha(x)$; thus the claim.

(saturation). Let $L \in Z(O, G, \sigma, x)$ and $L' \simeq L$. By the definition of causal equivalence, there exists a permutation $\tau \in \mathfrak{S}\mathbb{N}$ of the natural numbers, such that $L' = L\tau \simeq L$. Let S be the extension of L as above and $H \in O(G, \sigma, S)$ the history which outputs 1 at x in the last event of L involving x. Since O satisfies the property *(causality)*, we have $H\tau \in O(G, \sigma, S\tau)$. And since τ respects the causal constraints, the history $H\tau$ also outputs 1 at x in the last event of L involving x. Hence, $L\tau \in Z(O, G, \sigma, x)$.

(extension). We now show that for any finite schedules A and B, if $L \in Z(O, G, \sigma, x)$, then $A \cdot L \cdot B \in Z(O, G, \sigma, x)$. It suffices to prove that $A \cdot L$ and $L \cdot B$ belong to $Z(O, G, \sigma, x)$. In the following, S and H are the schedule and history associated with L as above.

We know $H \in O(G, \sigma, S)$ outputs 1 at x in the last event of L involving x. Since O satisfies *(unreliable delay)*, the history H' with schedule $A \cdot S$ which outputs 0 during A and the same values as H during S, belongs to $O(G, \sigma, A \cdot S)$. This history H' outputs 1 at x in the last event of $A \cdot L$; whence $A \cdot L \in Z(O, G, \sigma, x)$.

We now prove that $L \cdot B \in Z(O, G, \sigma, x)$. Consider any extension $S' \sqsupset L \cdot B$. Since O satisfies *(no future)*, there exists a history $H' \in O(G, \sigma, S')$ such that $H'|_L = H|_L$. In particular, H' outputs 1 at x in the last event of L involving x. Since O satisfies *(unreliable delay)*, it is possible to transform (by delaying the outputs of the history at x) H' into a history $H'' \in O(G, \sigma, S')$ which outputs 1 at x in the last event of $L \cdot B$ involving x. Thus, $L \cdot B \in Z(O, G, \sigma, x)$. □

Since $Z(O, G, \sigma, x)$ is closed under concatenation on the left or right, it is possible to define the notion of *minimal* schedule.

Definition 4 (Sets $Z_0(O, G, \sigma, x)$)**.** *A schedule $K \in Z(O, G, \sigma, x)$ is minimal if it cannot be written $K = A \cdot L \cdot B$ with $L \in Z(O, G, \sigma, x)$ in a non-trivial manner (i.e. A or B non-empty). The set of minimal schedule of $Z(O, G, \sigma, x)$ is denoted by $Z_0(O, G, \sigma, x)$.*

The following proposition shows that the family of minimal schedules satisfy the same properties as the family *Cones* in the definition of *Mnemosyne* (Sec. 5.3).

Proposition 15. *For every schedule $L \in Z(O, G, \sigma, x)$, there exists a minimal schedule $K \in Z_0(O, G, \sigma, x)$ such that K is a commuting factor of L. In addition, the family $\{Z_0(O, G, \sigma, x)\}$ of minimal schedules satisfy the following properties:*

- (anonymous) *for every permutation $\alpha \in \mathfrak{S}G$ of the agents, $K \in Z_0(O, G, \sigma, x)$ if and only if $\alpha K \in Z_0(O, G, \alpha\sigma, \alpha(x))$.*
- (cone) *every schedule in $Z_0(O, G, \sigma, x)$ is a past cone at x.*
- (saturation) *if $K \in Z_0(O, G, \sigma, x)$ and $K \simeq K'$, then $K' \in Z_0(O, G, \sigma, x)$*
- (unavoidable) *for every (weakly) fair schedule S, there exists an agent x such that S contains, as a commuting factor, a schedule from $Z_0(O, G, \sigma, x)$.*

Proof. The first claim is proven by a direct induction. The properties *(anonymous)* and *(saturation)* for Z_0 are direct consequences of the corresponding properties for Z.

We prove the property *(cone)*. Let $K \in Z_0(O, G, \sigma, x)$ be a minimal schedule, and p be the last event in K involving the agent x. Let L be the schedule corresponding to the causal past of p in K.[3] Then, we have $K \simeq L \cdot B$ for some schedule B.

Since $L \cdot B \simeq K\tau$ also belongs to $Z(O, G, \sigma, x)$, there exists an extension S of $K\tau$ and a history $H \in O(G, \sigma, S)$ that outputs 1 at x in the last event of $L \cdot B$ involving x. By construction, this last event occurs in L. Hence, $L \in Z(O, G, \sigma, x)$. The minimality of K implies that B is the empty schedule. Hence K is a past cone at x.

We now prove the property *(unavoidable)*. Let S be any weakly fair schedule. Let $H \in O(G, \sigma, S)$ be any legal history. By the property *(liveness)* of O, H eventually outputs 1 at some agent x in some event p during S. Therefore, the prefix $S \uparrow p$ belongs to $Z(O, G, \sigma, x)$. By the first claim, there exists a schedule $K \in Z_0(O, G, \sigma, x)$ which is a commuting factor of $S \uparrow p$. □

Before stating the last proposition, we need the following definition.

Definition 5 (Adherence)**.** *Let $\mathcal{H}$ be a set of histories. The* adherence of $\mathcal{H}$*, denoted by $adh(\mathcal{H})$, is the set of histories H such that, for any prefix L of its underlying schedule S, there exists a history $H^L \in \mathcal{H}$ with schedule S satisfying $H^L|_L = H|_L$.*

Given an oracle $O \in \mathcal{U}$, we define the oracles $\overline{O}$ and $\overline{O}^+$ as follows

$$\overline{O}(G, \sigma, S) = adh(O(G, \sigma, S)) \tag{5.4}$$

$$\overline{O}^+(G, \sigma, S) = adh(O(G, \sigma, S)) - \{\textit{zero history}\} \tag{5.5}$$

Intuitively, a history H belongs to the adherence of a set $\mathcal{H}$ of histories if one cannot determine, by looking at finite prefixes of H, if H actually belongs to the set $\mathcal{H}$ or not. We can now state the main proposition of this section.

Proposition 16. *Let $O \in \mathcal{V}$ and O^* be the oracle of Mnemosyne such that $Cones(O^*, G, \sigma, x) = Z_0(O, G, \sigma, x)$. Then*

$$\overline{O}^+ \preccurlyeq O^* \preccurlyeq O \tag{5.6}$$

Proof. We will prove that *(a)* O is a sub-behaviour of O^*, and *(b)* O^* is a sub-behaviour of $\overline{O}^+$.

(a). Let $H \in O(G, \sigma, S)$ be a legal history of O. If H outputs 1 at some agent x in some event p, then $S \uparrow p$ belongs to $Z(O, G, \sigma, x)$, and, thus, contains, as a commuting factor, some schedule from $Z_0(O, G, \sigma, x) = Cones(O^*, G, \sigma, x)$. In addition, by the liveness property of O, we know that H eventually outputs 1 at some agent in some event during S. Therefore, H is also a legal history of O^*. In other words, O is a sub-behaviour of O^*.

(b). Let $H^* \in O^*(G, \sigma, S)$. By the liveness property of O^*, we already know that H^* is not the zero history. It remains to show H^* is in the adherence of $O(G, \sigma, S)$. We prove it by recurrence. Assume we have already found, for some prefix $L \sqsubset S$, a history $H^L \in O(G, \sigma, S)$ such that $H^L|_L = H|_L$. Let's

[3] Many choices are available, but they are all causally equivalent.

write $S = L \cdot e \cdot \ldots$ where $e = (x, y)$ is the event occurring right after L. We will build a legal history H^{Le} of O out of H^L which matches with H on $L \cdot e$.

Let $v^* = (v^*_x, v^*_y)$ (resp. $v = (v_x, v_y)$) be the values output by H^* (resp. H^L) in e. We have four cases:

- $v^* = (0, 0)$: If $v_x = v_y = 0$, then there is nothing to do; taking $H^{Le} = H^L$ suffices. If $v_x = 1$, then by using the unreliable delay property of O, we can delay the values in H^L so that the resulting H^{Le} outputs 0 at x in e. Idem, if $v_y = 1$.
- $v^* = (0, 1)$: If $v = (0, 1)$, then there is nothing to do. If $v_x = 1$, the same technique as above is sufficient. The subtle case is $v_y = 0$. The fact that H^* outputs 1 at y in e implies that $S \uparrow e$ belongs to $Z(O, G, \sigma, y)$. Hence, there exists an extension $S' \sqsupset S \uparrow e$, and a history $H' \in O(G, \sigma, S')$ which outputs 1 at y in e. Using the schedule dependent property of O, we can transform H^L to output 1 at y in e; the resulting history H^{Le} is also a legal history of O.
- $v^* = (1, 0)$ or $v^* = (1, 1)$: : The same reasoning as above yields the claim.

Hence, we managed, in every case, to build a legal history H^{Le} of O which matches with H on the prefix $L \cdot e$. To start the induction, it suffices to take the empty schedule for L. Therefore, H^* is in the adherence of $O(G, \sigma, S)$. □

Intuitively, the oracles O and $\overline{O}^+$ are both live (each of them eventually outputs 1 at some point), and cannot be distinguished by looking at finite prefixes. Therefore, if the studied task involves a notion of termination, there is a chance that O solves this task if and only if $\overline{O}^+$ solves this task.

Conjecture 1. *A "terminating" problem P (such as consensus) can be solved using the oracle O if and only if it can be solved using the oracle $\overline{O}^+$.*

If this conjecture is true (which would require to define formally what a terminating problem is), then Prop. 16 implies that oracles in $\mathcal{V}$ and oracles in *Mnemosyne* solves the same set of "terminating" problems.

Chapter 6

Leader Election

6.1 Introduction

Leader election, like consensus, is a fundamental problem in distributed computing. This task mainly consists in selecting a unique agent in the system, and turns out to be impossible to solve in many cases. The impossibility is usually related to the system asynchrony, limited resources, the presence of failures, their type, or other general conditions.

Actually, the leader election problem is intimately related to the consensus problem. Indeed, as already explained in Chap. 5, [47] have shown that the consensus is impossible in asynchronous message-passing systems where one processor may crash, and [32] have circumvented this issue by introducing the notion of failure detectors. Among the different failure detectors proposed to solve consensus in the conventional asynchronous communication model, the *eventual leader elector* Ω, has been proven to be the *weakest* [31]. Informally, that means that it supplies the minimum supplementary information necessary to obtain a solution.

In this chapter, we mainly study the *self-stabilizing leader election* ($SSLE$) problem in population protocols. Due to the harsh constraints of population protocols, it is not surprising that this problem is impossible in many cases [10, 46, 12]. Self-stabilization [39] is a framework for dealing with transient state-corrupting faults and can be viewed as allowing the system to start from an arbitrary configuration. In other words, a protocol solves a problem in a self-stabilizing way if every feasible execution starting from any initial configuration solves the problem.

The eventual leader elector Ω of Chandra and Toueg and other classical failure detectors cannot be used with population protocols, because they assume that the network nodes have unique identifiers, unavailable to anonymous agents in population protocols. Many other previous oracles, like those proposed for anonymous models (e.g., [22]), cannot be used in population protocols either, because of the memory constraints imposed by the model (this issue is discussed in Chap. 5, Sec. 5.1).

To deal with this issue, Fischer and Jiang introduced a new type of oracle, called the *eventual leader detector* [46] and denoted by $\Omega?$. Instead of electing a leader, like Ω, $\Omega?$ simply reports to each agent an (eventually correct) estimate about whether or not one or more leaders are present in the network (see

Sec. 6.4 for a formal definition). This oracle does not require unique identifiers and has additional drastic differences. One of the important differences is motivated by the self-stabilizing nature of the $SSLE$ problem considered in [46].

While Ω is designed to circumvent impossibility related to crash faults, $\Omega?$ is designed to deal with state-corrupting faults. Thus, while Ω is related to a failure pattern and is independent of the protocol using it, $\Omega?$ interacts with the protocol, providing information related to the system configurations reached during the execution. With $\Omega?$, there is some sort of feedback loop: the outputs of the oracle influence the protocol; and conversely, the protocol influences the outputs of the oracle. Yet, there are some features in common with Ω. Both Ω and $\Omega?$ are unreliable in the sense that $\Omega?$ can make errors, that is, to give false information at some point and at some agents, and is only required to eventually provide correct answers, similarly to Ω. Finally, such weak guarantees allow both Ω and $\Omega?$ to be implemented in practice using timeouts and other features often found in real systems (more details about the implementation of $\Omega?$ can be found in [46]; about Ω, in [32]).

A part of this work has been published in [12].

Related Work

Being an important primitive in distributed computing, leader election has been extensively studied in various other models, however much less in population protocols. Because of model differences, previous results do not directly extend to the model considered here. For surveys on these previous results in other models, refer to [10, 46]. In the following, we mention only the most relevant works to $SSLE$ in population protocols.

It was shown, e.g. in [8, 16], that fast converging population protocols can be designed using an initially provided unique leader. Moreover, many self-stabilizing problems on population protocols become possible given a leader (though together with some additional assumptions, see, e.g., [10, 14]). Nevertheless, $SSLE$ is impossible in population protocols over general connected communication graphs [10]. Yet, [10] presents a non-uniform solution for $SSLE$ on rings. A uniform algorithm for rings and complete graphs is proposed in [46], but uses $\Omega?$. Recently, [26] showed that at least n agent states are necessary and sufficient to solve $SSLE$ over a complete communication graph, where n is the population size (unavailable in population protocols). For the enhanced model of *mediated population protocols* (MPP) [64], it is shown in [66] that $(2/3)n$ agent states and a single bit memory on every agent pair are sufficient to solve $SSLE$. It is also shown that there is no MPP that solves $SSLE$ with constant agent's state and agent pair's memory size, for arbitrary n. In [27], versions of $SSLE$ are considered assuming $\Omega?$ together with different types of *local fairness* conditions, in contrast with the original population protocols' *global fairness*.

The Problem

We formally define the behaviour $\mathcal{ELE}$ corresponding to the leader election problem. $\mathcal{ELE}$ is defined with the input alphabet $\{\perp\}$ (i.e., no input) and the output alphabet $\{0, 1\}$ such that, given a graph G and a schedule S on G, a

history $H \in \mathcal{ELE}(G, S)$ if and only if its associated trace T has a constant suffix $T' = \alpha\alpha\alpha\ldots$ and there exists an agent λ such that $\alpha(\lambda) = 1$ and $\alpha(u) = 0$ for every $u \neq \lambda$. In other words, λ is eventually permanently the unique leader. Note that for all our protocols, there is an implicit output map that maps a state to 1 if it is a leader state, and to 0 otherwise.

In our framework, the informal problem of Self-Stabilizing Leader Election ($SSLE$) consists in obtaining a population protocol that solves $\mathcal{ELE}$ using another behaviour (if necessary) and starting from arbitrary initial configurations.

Note that, in contrast to some formulations, the agents are not required to know when a leader is elected. Put another way, there is no termination condition in this formulation. This stems from the fact that a self-stabilizing solution to a one-shot problem is meaningless[1].

Contexts and Overview

We will study the leader election problem in several contexts. Recall that a context is defined by a family of communication graphs, an initialization procedure and a fairness condition. The contexts used in this chapter are summarized in the following table:

Context	Graph family $\mathcal{F}$	Initialization	Fairness	Sections
(1)	Contains a covering	Uniform	Local	6.2
(2)	Arbitrary	Uniform	Global	6.3
(3)	Rings	Arbitrary	Global	6.5
(4)	Bounded-degree	Arbitrary	Global	6.6
(5)	Arbitrary	Arbitrary	Global	6.7, 6.8
(6)	Non-simple family	Arbitrary	Global	6.9

We first start by examining the problem in a non self-stabilizing setting. The contexts, (1) and (2), deal with uniformly initialized protocols. A uniform initialization means that, in every execution, all the agents start with the same initial state. In Sec. 6.2, it is shown that, in the context (1) no population protocol can implement leader election problem over a family of graphs which contains a covering (see Sec. 6.2 for details). On the hand, if we use the global instead of the local fairness, as in the context (2), then the leader election becomes solvable over arbitrary graphs. This result highlights the power of global fairness.

In the following contexts, (3) to (6), from Sec. 6.4 until the end of the chapter, we will focus on self-stabilization, and assume everywhere that the considered protocols start in arbitrary initial configurations. All these contexts impose the global fairness, and differ only in the graph family. We then speak of *self-stabilizing* protocols, solutions, or implementations, to highlight the fact that the corresponding context assumes an arbitrary initialization.

In (almost) all these contexts, it is proven that the leader election problem admits no self-stabilizing implementations. In Sec. 6.4, we present the main reason (drawn from [10]) for this impossibility. We also introduce a new class of oracles to circumvent this issue that generalize Fischer and Jiang's oracle $\Omega?$ [46]. Note that, as explained in the introduction, in contrast with Chap. 5, these

[1] We will come back to this point in the second part of this thesis where we examine the idea of a self-stabilizing replicated state-machine.

oracles do observe the outputs of the protocols with which they are composed (non-trivial input alphabet).

Fischer and Jiang have proposed a self-stabilizing solution to the leader election problem using $\Omega?$ over the family of rings. In Sec. 6.5, we show that the leader election problem is actually equivalent to the oracle $\Omega?$ (as an equivalence of behaviours). In particular, this shows that any oracle strong enough to yield a self-stabilizing solution to the leader election on rings, is in fact stronger than $\Omega?$.

In Sec. 6.6, we propose a self-stabilizing solution to leader election using the same oracle $\Omega?$ over the family of bounded-degree graphs (with a known upper bound). Next, we study the same problem over the more general family of arbitrary graphs; this requires, a priori, oracles stronger than $\Omega?$. In Sec. 6.7, we provide a simple solution, using the oracle $\Omega?(2)$ from the class defined in Sec. 6.4. In Sec. 6.8, we provide a more intricate solution using a (a priori) weaker oracle, namely $\Omega? \otimes \Omega?$ (the parallel composition of two copies of $\Omega?$), over arbitrary graphs.

Finally, in Sec. 6.9, we show that, in contrast to the equivalence of $\Omega?$ and leader election on rings, these problems are not equivalent over any non-simple family (see Sec. 6.9 for details). This implies that there is no self-stabilizing implementation of $\Omega?$ using the leader election behaviour over, for instance, the family of complete graphs, or arbitrary graphs with bounded degree, and many more.

Sec. 6.10 develops a technical issue that is used in many of the previous sections.

6.2 Impossibility with Local Fairness, Uniform Initialization

In this section, we show that the eventual leader election problem cannot be solved by any uniformly initialized population protocol under the local fairness assumption. A population protocol is uniformly initialized if there is unique initial state for the agents, i.e., an initial configuration assigns the same state to every agent.

We first recall the notion of *graph covering* [5, 20]. A *fibration* (resp. *opfibration*) between graphs G and B is a graph morphism $\phi : G \to B$ such that for every node b in B, for every node y satisfying $\phi(y) = b$, ϕ induces a bijection between the set of incoming (resp. outgoing) edges at y and the set of incoming (resp. outgoing) edges at b. A *covering* from G to B is a graph morphism from G to B that is both a fibration and an opfibration. The graph G is called the *total* graph, and B is the *base* graph. The *fiber* over a node b in B is the set of nodes in G that are mapped to b via ϕ, which we denote by $\phi^{-1}(b)$. A fiber is *trivial* if it is a singleton. A covering is a *k-covering* if every fiber has k elements, i.e., $\forall b, |\phi^{-1}(b)| = k$. For instance, there is a covering from a ring of size $2 \cdot n$ to a ring of size n obtained by mapping two diametrically opposite nodes to the same node.

The following proposition is inspired by the impossibility result of leader election in the family of rings under local fairness [46] and the ideas developed in [5, 20]. Note that the models considered in [5, 20] are different from the population protocols. Hence, the results do not directly apply to our case.

Proposition 17. *Let $\mathcal{F}$ be a family of graphs that contains graphs G and B such that there exists a k-covering $\phi : G \to B$ with $k \geq 2$. There is no uniformly initialized population protocol that implements $\mathcal{ELE}$ over the family $\mathcal{F}$ under the local fairness assumption.*

Proof. We prove the result by contradiction. Assume that there exists a protocol A that solves the leader election problem with uniform initialization (all agents are initially in the same state q) under local fairness. We first show how to simulate a step of A on B with a specific sequence of steps on G. Then we show how to lift any locally fair execution on B to a locally fair execution on G, and finally we prove the contradiction.

(Simulation). Consider configurations γ, γ' on B and an action $\sigma = ((a, b), (p, q) \to (p', q'))$ enabled in γ such that $\gamma \xrightarrow{\sigma} \gamma'$. Since ϕ is an opfibration, we know that for each node x_i in $\phi^{-1}(a)$ $(1 \leq i \leq k)$, there is a unique edge (x_i, y_i) that is mapped to (a, b); then let $s_i = ((x_i, y_i), (p, q) \to (p', q'))$ be an action (on G). If there were indices $i \neq j$ such that $y_i = y_j = y$, then y would have two incoming edges that are both mapped to the edge (a, b); whence a contradiction with the fact that ϕ is a fibration. Hence, the y_i's are pairwise distinct (as well as the x_i's by definition).

We denote by u_0 the configuration on G such that $u_0(\phi^{-1}(c)) = \{\gamma(c)\}$ for every c in B. The action s_1 is enabled in u_0 since $(u_0(x_1), u_0(y_1)) = (\gamma(a), \gamma(b)) = (p, q)$. Thus the configuration u_1 such that $u_0 \xrightarrow{s_1} u_1$ is well-defined, and we have $(u_1(x_0), u_1(y_0)) = (p', q')$. The action s_2 is enabled in u_1 since $x_1 \neq x_2$, $y_1 \neq y_2$ and (thus) $(u_1(x_1), u_1(y_1)) = (u_0(x_1), u_0(y_1)) = (p, q)$. Hence, the configuration u_2 such that $u_1 \xrightarrow{s_2} u_2$ is well defined. We can iterate the construction until $i = k$. In the last configuration we have $(u_k(x_i), u_k(y_i)) = (p', q')$ for every $1 \leq i \leq k$. Actually, $u_k(\phi^{-1}(b)) = \{\gamma'(b)\}$ for every agent b in B. In other words, we have simulated the step $\gamma \to \gamma'$ in B by a sequence of steps $u_0 \xrightarrow{*} u_k$ in G.

(Locally Fair Lift). Consider a locally fair execution $E_B = \gamma_0\gamma_1 \dots$ of A on the graph B; we have $\forall b, \gamma_0(b) = q$. Thanks to the simulation above, we can build a virtual execution $E_G = g_0 \dots g_1 \dots g_2 \dots$ of A on G such that for every $t \in \mathbb{N}$, for every node $b \in B$, $g_t(\phi^{-1}(b)) = \{\gamma_t(b)\}$. Note that g_0 maps every node in G to q, so E_G is uniformly initialized.

We show that E_G is locally fair. Assume that an action $s = ((x, y), (p, q) \to (p', q'))$ is enabled infinitely often in E_G. The construction of E_G involves that s is enabled in g_i for infinitely many i. But, since $(g_i(x), g_i(y)) = (p, q) = (\gamma_i(\phi(x)), \gamma_i(\phi(y)))$, the action $\sigma = ((\phi(x), \phi(y)), (p, q) \to (p', q'))$ is enabled infinitely many times in E_B. Hence, by local fairness, there are infinitely many i such that $\gamma_i \xrightarrow{\sigma} \gamma_{i+1}$. Then, for infinitely many i, the construction of the sequence $g_i \xrightarrow{*} g_{i+1}$ involves that the action s is triggered during it. Whence E_G is locally fair.

(Contradiction). If A solves the leader election problem, there exists some $i_0 \in \mathbb{N}$ such that for every $i \geq i_0$, the configuration γ_i on B outputs a unique leader at λ. By construction, for every $l \in \phi^{-1}(\lambda)$, $g_i(l) = \gamma_i(\lambda)$. This involves that g_i outputs a leader at k agents (since $|\phi^{-1}(\lambda)| = k$) for infinitely many i. This contradicts the fact that any locally fair execution of A solves leader election on G.

Note that imposing only that ϕ is a fibration (or an opfibration) is not enough to lift a locally fair execution on the base graph to a locally fair execution on the total graph. □

6.3 Solution with Global Fairness, Uniform Initialization

We establish that, under global fairness, solving the leader election problem on arbitrary communication graphs is possible without oracle, when an uniform initialization is possible (Alg. 6). In other words, there exists a uniformly initialized population protocol that solves the $\mathcal{ELE}$ problem over the family of all graphs under the global fairness assumption. This result highlights the difference between global and local fairness. It also shows that the necessity to use an oracle comes from the requirement of self-stabilization. We focus on *strongly connected* graphs; Sec. 6.10 shows how to extend these results to weakly connected graphs. Each agent x can be leader or non-leader (implemented with a variable $leader_x$) and can hold a white or black token (implemented with a variable $token_x$). Initially, every agent is a leader and holds a black token (uniform initialization). The tokens move through the network by swapping between two agents during an interaction. When two black tokens meet, one of them turns white. When a white token interacts with a leader x, x becomes a non-leader and the token is destroyed.

Algorithm 6: Leader Election with Uniform Initialization

```
1  variables for every agent x:
2      leader_x : 0 (non-leader) or 1 (leader);
3      token_x : ⊥ (no token), white or black;
4  initialization: ∀x, (leader_x, token_x) = (1, black);          /* uniform */
5  protocol (initiator x, responder y):
6      if token_x = token_y = black then
7          token_y ← white;
8      if token_x = white ∧ leader_y = 1 then
9          leader_y ← 0 ;                    /* y becomes a non-leader */
10         token_x ← ⊥ ;                     /* the token is destroyed */
11     token_x ↔ token_y;                          /* swap the tokens */
```

We consider an execution E of Alg. 6 and prove that there is eventually a unique leader. We will use the formalism of traces instead of histories, and the outputs (leader or not leader) are encoded in the states. Hence an execution E is represented by a sequence of configurations and input assignments $(\gamma_t, \alpha_t)_{t\in\mathbb{N}}$. We denote by E_∞ the infinite suffix of E such that each couple (γ, α) in E_∞ occurs infinitely often in E_∞. Given a configuration γ, let $b(\gamma)$ be the number of black tokens, $w(\gamma)$ the number of white tokens and $l(\gamma)$ the number of leaders in γ. In addition, for every agent x, we denote by $\gamma.leader_x$ (resp. $\gamma.token_x$) the value of the variable $leader_x$ (resp. $token_x$) in the configuration γ.

Lemma 7. *In each configuration γ in every execution E of Alg. 6, $b(\gamma) + w(\gamma) = l(\gamma)$ and $b(\gamma) \geq 1$.*

Proof. In the initial configuration, $b(\gamma) = l(\gamma) = n$ the number of agents, and $w(\gamma) = 0$. We show that for any configuration γ satisfying the property, any configuration γ' such that $\gamma \to \gamma'$, γ' satisfies the property. In the algorithm,

the swapping of tokens (line 11) does not modify the number of tokens nor the number of leaders. If line 7 is executed, then $b(\gamma') = b(\gamma) - 1 \geq 1$ (the condition in the if statement implies $b(\gamma) \geq 2$), $w(\gamma') = w(\gamma) + 1$ and $l(\gamma') = l(\gamma)$; whence $b(\gamma') + w(\gamma') = l(\gamma')$. If lines 9 and 10 are executed, then $b(\gamma') = b(\gamma) \geq 1$, $w(\gamma') = w(\gamma) - 1$ and $l(\gamma') = l(\gamma) - 1$; whence $b(\gamma') + w(\gamma') = l(\gamma')$. Hence, in all cases, γ' also satisfies the property. □

Lemma 8. *For every configuration γ in E_∞, $b(\gamma) = 1$.*

Proof. First note that, since no black token is ever created in Alg. 6, if $\gamma \to \gamma'$, then $b(\gamma) \geq b(\gamma')$. Hence, the number of black tokens cannot increase during E_∞. Assume that there is a configuration γ in E_∞ such that $b(\gamma) = t \geq 2$. By global fairness, there is a configuration in E_∞ where two black tokens are in two neighboring nodes. From this configuration, there is a reachable configuration γ' resulting from the interaction of these two neighbors. In γ', $b(\gamma') \leq t - 1 < b(\gamma)$. The global fairness ensures that γ' is in E_∞. By the first remark, γ cannot occur in E_∞ after the first occurrence of γ'. This is a contradiction with the definition of E_∞. □

Proposition 18. *In ever execution E of Alg. 6, there exists exactly one agent λ such that for every configuration γ in E_∞, $\gamma.leader_\lambda = 1$ and for every agent $\mu \neq \lambda$, $\gamma.leader_\mu = 0$.*

Proof. We show by contradiction that for every γ in E_∞, $w(\gamma) = 0$. Assume that there exists a γ such that $w(\gamma) \geq 1$. Since $b(\gamma) = 1$, $l(\gamma) = w(\gamma) + b(\gamma) = w(\gamma) + 1 \geq 1$. By global fairness, there is a configuration in E_∞ where a white token and a leader are in two neighbouring nodes. From this configuration, there is a reachable configuration γ' resulting from the interaction of these two neighbours such that $l(\gamma') < l(\gamma)$. The global fairness ensures that γ' is in E_∞. Since γ is also in E_∞, there must be a sequence of steps $\gamma' \xrightarrow{*} \gamma$. During this sequence, a leader must be created. This is impossible since no leader is ever created. Then, $w(\gamma) = 0$ for every γ in E_∞. This implies that $l(\gamma) = w(\gamma) + b(\gamma) = 0 + 1 = 1$ for every γ in E_∞. Since the variables $leader_x$'s are never swapped, there exists an agent λ such that for every configuration γ in E_∞, $\gamma.leader_\lambda = 1$ and for every agent $\mu \neq \lambda$, $\gamma.leader_\mu = 0$. □

6.4 Oracles $\Omega?(d)$

Motivation

Let's first define the notion of non-simple family of graphs (introduced in [10]).

Definition 6 (Non-simple Graph Family)**.** *A graph family $\mathcal{F}$ is* non-simple *if there are exist graphs $G, G_1, G_2 \in \mathcal{F}$ such that G_1 and G_2 are two disjoint subgraphs of G. If there are no such graphs, the family is* simple*.*

For instance, the family of complete graphs is non-simple, whereas the family of rings is simple. In [10], the authors have shown that, even with the global fairness, if the graph family $\mathcal{F}$ is non-simple, then there is no self-stabilizing solution to the leader election problem over $\mathcal{F}$. The argument relies on a well-known partitioning technique.

In the same paper [10], the authors have managed, for each $k \geq 2$, to propose a self-stabilizing solution over any family of rings whose sizes are not multiples of k. Yet, it is still unknown whether there exists a self-stabilizing solution over the whole family of rings. We conjecture that such a solution does not exist.

Conjecture 2. *There is no protocol that implements $\mathcal{ELE}$ over the rings, with global fairness, and arbitrary initialization.*

In [46], the authors have introduced the oracle $\Omega?$ which somehow gives information about the absence of leaders in the system. Using $\Omega?$, they proposed a self-stabilizing solution over the whole family of rings. In this section, we define a class of oracles that generalize $\Omega?$.

Definition

We define, for each $d \geq 1$, an oracle $\Omega?(d)$. Its input alphabet is $\{0, 1\}$, and its output alphabet is $\{0, \ldots, d\}$. The domain of $\Omega?(d)$ is all the graphs.

For sake of simplicity, we define the oracle $\Omega?(d)$ in terms of traces, instead of histories. Recall the two formulations are equivalent (see Chap. 3, Sec. 3.1).

Intuitively, the oracle observe its input history, and gives an estimate (up to the maximum value d) of the number of agents that are assigned[2] the value 1.

Given an assignment α, we denote by $l(\alpha)$ the number of vertices that are assigned the value 1 by α. If $0 < r \leq (\alpha) \leq r' \leq d$ for all α in a suffix of the input trace (i.e. trace associated with the input history), then the oracle will eventually permanently output values in $\{r, \ldots, r'\}$ at every agent. When $l(\alpha) = 0$ for all α in a suffix the input trace, it is only required that the oracle permanently outputs 0 at some agent (at least one).

More formally, let H_{in} (resp. H_{out}) be a history with values in $\{0, 1\}$ (resp. $\{0, \ldots, d\}$, and T_{in} (resp. T_{out}) its associated trace. The histories H_{in} and H_{out} have the same underlying schedule. Then $H_{out} \in \Omega?(d)(G, H_{in})$ if and only if the traces T_{in} and T_{out} satisfy the following conditions:

- If T_{in} has a suffix which is uniform constant with value 0, then T_{out} has a suffix in which at least one agent is permanently assigned the value 0.
- For every $1 \leq r \leq r' \leq d$, if T_{in} has a suffix $\alpha_0\alpha_1\ldots$ such that $\forall s, r \leq l(\alpha_s) \leq r'$, then T_{out} has a suffix with values in the interval $\{r, \ldots, r'\}$.
- If T_{in} does not match any of the previous conditions, then any T_{out} is possible.

Note that $\Omega?(1)$ corresponds to the Fischer and Jiang's oracle $\Omega?$ in [46]. It is easy to see that the oracles $\Omega?(d)$ are linearly ordered: if $d \leq d'$, then $\Omega?(d')$ is a sub-behaviour of $\Omega?(d)$, hence $\Omega?(d) \preccurlyeq \Omega?(d')$ (in any context).

[2] These agents are usually referred to as leaders, but, at this stage, it is just a convention.

6.5 Equivalence of $\mathcal{ELE}$ and $\Omega?$ over Rings

Thanks to [46], we already know that $\Omega?$ is stronger than $\mathcal{ELE}$ over rings. One may then wonder how strong $\Omega?$ is. We answer this question by showing that $\Omega?$ and $\mathcal{ELE}$ are actually equivalent over rings. Precisely, we define the *RingDetector* protocol (see Algorithm 7) that uses the output of $\mathcal{ELE}$ to implement $\Omega?$. We also assume that the rings are oriented, since the authors in [10] have presented a self-stabilizing ring orientation protocol.

For the sake of clarity, the unique leader provided by $\mathcal{ELE}$ is called the *master*, whereas the output of $\Omega?$ reports about the *leaders*. Hence, the goal consists in the master detecting the presence or the absence of leaders in the graph, that is to mimic $\Omega?$.

RingDetector

Let us define the self-stabilizing protocol *RingDetector*. The input variables (read-only) at each agent x are: the *master bit* $master_x$ (values in $\{0,1\}$) that keeps the output of $\mathcal{ELE}$; and the *leader bit* $leader_x$ (values in $\{0,1\}$), which represents the input of $\Omega?$. The working variables are: the *probe field* $probe_x$ (with values: $\perp$ - no probe, or 0 - white probe, or 1 - black probe); the *token field* (with values: $\perp$ - no token, or 0 - white token, or 1 - black token); the *flag bit* $flag_x$ (with values: 0 - cleared, 1 - raised); and the *output bit* (values in $\{0,1\}$), which represents the corresponding output of $\Omega?$.

Each time an agent has its leader bit set to 1, it raises its flag (and the flag of the other agent in the interaction) – line 5. A token moves clockwise, and its purpose is to detect a leader (actually, a raised flag) and to report it to the master (lines 18 –26). A probe moves counter-clockwise, and its purpose is to report to the master the lack of tokens (lines 7 – 13). The master loads a white probe each time it is the responder of an interaction (line 2). When a probe meets a token, the probe becomes black (line 10). When two probes meet, they merge into a black probe if one of them was black, and into a white probe otherwise (line 12). The master loads a token colored with its flag only when it receives a white probe (line 17). Each time a token meets an agent with its flag raised, the token becomes black (line 21) and the flag is cleared (line 25). Two meeting tokens merge into a black token if one of them is black, and into a white token otherwise (line 23). When the master receives a token, it whitens the token, and it outputs 0 if the token is white, and 1 otherwise (lines 28 –31). In any interaction, the responder copies the output of the initiator, unless the responder is the master (line 33).

Correctness

In the following, the input trace $T = \alpha_0\alpha_1\ldots$ of every execution E is assumed to provide a unique master, i.e., there exists a unique agent λ in E such that $\alpha_i(\lambda).master = 1$ for all i.[3] By the definition of $\mathcal{ELE}$ and *RingDetector*, such an input trace exists in an infinite suffix of every E of *RingDetector*. For the correctness proof, we focus only on such suffixes, for every execution.

[3]We precise the notations. α being an assignment (resp. C a configuration), $\alpha.v$ (resp. $C.v$) is the projection of α (resp. C) on the variable v; and $\alpha(x).v$ (resp. $C(x).v$) is the value of this projection at agent x.

Algorithm 7: Protocol *RingDetector* - initiator x, responder y

```
1  (if the master is the responder, it creates a white probe);
2  if master_y = 1 then probe_y ← 0;
3  ;
4  (raise flags if needed);
5  if leader_x ∨ leader_y then flag_x ← flag_y ← 1;
6  ;
7  (move probe from y to x);
8  if probe_y ≠ ⊥ then
9      (the probe becomes black when meeting a token);
10     if token_x ≠ ⊥ then probe_x ← 1;
11     otherwise, keeps the same color or merges);
12     else if probe_x ∈ {⊥, 0} then probe_x ← probe_y;
13     probe_y ← ⊥;
14 end
15 ;
16 (if the master receives a white probe, it loads a token);
17 if master_x = 1 and probe_x = 0 then token_x ← flag_x;
18 (move token from x to y);
19 if token_x ≠ ⊥ then
20     (the token becomes black when meeting a flag);
21     if flag_y = 1 then token_y ← 1;
22     (otherwise, keeps the same color or merges);
23     else if token_y ∈ {⊥, 0} then token_y ← token_x;
24     (the flag is cleared);
25     flag_y ← 0;
26     token_x ← ⊥;
27 end
28 (if the master receives a token, it changes its output and whitens the token);
29 if master_y = 1 and token_y ≠ ⊥ then
30     out_y ← token_y;
31     token_y ← 0;
32 (a non-master responder copies the output of the initiator);
33 if master_y = 0 then out_y ← out_x;
```

The leader bit component in the input trace corresponds to the input of $\Omega?$. In this trace, leaders can appear or disappear almost freely, during each meeting. In particular a leader can "jump" from u to v during an interaction between u and v. Though, a leader cannot "jump" to a distant (non interacting) agent on the ring, by the compatibility of an input trace with a schedule (see Chap. 3, Sec. 3.1). The fact that a leader can "jump" counter-clockwise from the responder to the initiator introduce some subtleties in *RingDetector*. Without taking care, such a "jumping" leader could be undetectable. To ensure its detection, the flag bits of both the responder and the initiator are raised, even if the leader is detected only at one of the two interacting agents (line 5).

We use the following notations. Given an execution E, E_∞ denotes the maximum (infinite) suffix of E such that each couple (γ, α) (γ being a configuration, and α an input assignment) in E_∞ occurs infinitely often. IR_E denotes the (finite) set of configurations occurring in E_∞, i.e., the set of "infinitely recurrent" configurations.

Lemma 9. *For any execution E, in any configuration of IR_E, there is a unique agent holding a token (black or white).*

Proof. Consider a configuration $\gamma \in IR_E$. We first prove that in γ at least one agent holds a token. By contradiction, assume that, for every agent x, $\gamma(x).token = \bot$. The following scenario will produce a token. First, the master λ interacts as a responder and produces a white probe at λ. Then, all the other probes move (counter-clockwise) to the master. Then the white probe at λ visits all agents and returns to λ. Since there are no tokens in the graph, the white probe does not turn black. Then, the white probe arriving at λ produces a token (line 17). This scenario does not depend on the presence of leaders. Hence, there exists a configuration γ' with at least one token such that

$\gamma \xrightarrow{*} \gamma'$, for any input trace. By global fairness, $\gamma' \in IR_E$. Together with that, no rule of the protocol can remove all tokens. In line 26, the token is removed from an initiator x, but is present or created in the responder y (line 23). No other instruction removes a token. Thus γ cannot occur infinitely often; hence a contradiction. Hence, in γ at least one agent holds a token.

Assume now that γ has at least two tokens. Since two meeting tokens merge into one, there is a configuration γ' with exactly one token such that $\gamma \xrightarrow{*} \gamma'$, for any input trace. By global fairness, γ' belongs to IR_E. Since γ also occurs infinitely often in the execution, $\gamma' \xrightarrow{*} \gamma$, for any input trace. To reach γ, a token should be created. It can happen only if the master receives a white probe. Thus, the master should receive infinitely many white probes during E_∞. However, once there is a token, since the tokens move clockwise and the probes counter-clockwise, any probe arriving at the master must be black; hence a contradiction. Therefore, γ has exactly one token. □

Thus, in the suffix E_∞, there is a unique token moving clockwise. We divide E_∞ into *rounds*, defined as follows. A *round* begins with an interaction in which the master holds the token and is the initiator; the round ends with the first event in which the master is the responder and the initiator holds the token. In other words, a round corresponds to the token traveling around the whole ring starting and ending at the master.

Lemma 10. *Let R be a round in E_∞. We denote by $(\gamma_0, \alpha_0) \dots (\gamma_r, \alpha_r)$ the sequence of configurations and input assignments corresponding to R.*

case (a). *If there are no leaders in R (i.e., for every $0 \leq i \leq r$, and every agent x, we have $\alpha_i(x).leader = 0$), then after the last action in R, all the agents have their flags cleared (set to 0).*

case (b). *If there are no leaders in R, and if all agents have their flags cleared at the beginning of the round, then at the end of the round, the master outputs 0 and all agents have their flags cleared.*

case (c). *If there is at least one leader at each assignment α_i during the round, i.e., for every $0 \leq i \leq r$ there is some agent x_i such that $\alpha_i(x_i).leader = 1$, then at the end of the round, the master outputs 1.*

Proof. case (a). Assume there are no leaders during the round R. Since the token moves clockwise from the master to the master, and since a token clears any flag it encounters, at the end of the round, the token has cleared all the possible raised flags in the ring.

case (b). Assume that there are no leaders during R, and that all the flags are cleared at the beginning. During the first action in R, the master holds the token and colors it in white. Since there are no leaders in R, in every configuration within the round, all the flags are cleared. Hence, when moving clockwise from the master to the master, the token meets no raised flags and stays white. At the end of the round, the master receives a white token and outputs 0.

case (c). Assume that there is a leader at each assignment during the round. Let μ be a leader agent in the assignment α_0, i.e., $\alpha_0(\mu).leader = 1$. During the round, there must be some action i, such that $\mu = v_i$ is the responder, and the initiator u_i holds the token. If μ is a leader in an assignment α_i, then after the transition, the token turns black. If μ is not a leader, in assignment α_i, since μ is a leader in the assignment α_0, there must be some

action $j < i$ such that $\alpha_j(\mu).leader = 1$ and $\alpha_{j+1}(\mu).leader = 0$. Now, since the input trace is compatible with the schedule, μ must be the initiator u_j or the responder v_j in the transition $(\gamma_j, \alpha_j) \to \gamma_{j+1}$. Hence, μ must raise a flag in both the responder v_i and the initiator u_i (line 5), i.e., we have $\gamma_{j+1}(\mu).flag = 1$ $(j + 1 \leq i)$. Recall that there is a unique token, so the flag cannot be cleared during the remaining actions until i. Hence, at action i, the token turns black (line 21) when the token moves from the initiator u_i to the responder $v_i = \mu$. In all cases, the master receives a black token at the end of the round, and thus outputs 1. □

Proposition 19. *The protocol RingDetector is a self-stabilizing implementation of* $\Omega?$ *using* $\mathcal{ELE}$ *(i.e.,* $\Omega? \preccurlyeq \mathcal{ELE}$*) over oriented rings.*

Proof. Consider a globally fair execution E and focus on the suffix E_∞. By Lemma 16, in E_∞, there is a unique token moving clockwise. Let $E_\infty = \ldots R_1 R_2 \ldots R_i \ldots$, where each R_i is a round.

Consider first the case where the input trace $T = \alpha_0 \alpha_1 \ldots$ in E_∞ permanently assigns no leader everywhere, i.e., for every i, for every agent x, $\alpha_i(x).leader = 0$. By Lemma 10, at the end of R_1, all flags are cleared. Hence, at the end of R_2, the master outputs 0 and all flags are cleared. By iteration, at the end of each round R_i, $i \geq 2$, the master outputs 0. Since the master updates its output only when it receives the token, and since this happens exactly at the end of a round, in the suffix $R_2 R_3 \ldots$, the master permanently outputs 0. The fact that the responder always copies the output of the initiator (unless the responder is the master) implies that there is a suffix during which all agents permanently output 0.

Assume now that the input trace in E_∞ is such that there is at least one leader at every input assignment. By Lemma 10, at the end of each R_i, the master outputs 1. The same argument as above shows that there is a suffix of execution during which all agents permanently output 1.

Note that, in the remaining cases of input traces in E_∞, that is when there are input assignments with a leader and some other without, nothing has to be proven, because then, the output of $\Omega?$ is arbitrary. □

Remark 1. *Note that a simpler solution managing only tokens, sent periodically by the master, and without managing any probes, would not be correct. To see this, consider an input trace where there is one leader in every input assignment, but this leader moves repeatedly clockwise, "jumping" from one agent to its successor on the ring. By the definition of* $\Omega?$*, in this scenario, the master should eventually and permanently output* 1*. However, it is infinitely often possible that there are two tokens directly following the leader one after the other, during the whole tour, from the master to the master. In this case, the first token arriving at the master is black, but the following token is white. This is because the first token has cleared every flag raised by the leader. The repetition of this scenario causes an oscillation of the output of the master between* 0 *and* 1*.*

6.6 SSLE with Ω? over Bounded-Degree Graphs

In [46], the authors have proven that Ω? is strong enough to yield a self-stabilizing implementation of $\mathcal{ELE}$. In this section, we extend their result to the family of graphs with bounded degree. Precisely, given any integer d, the behaviour $\mathcal{ELE}$ can be implemented using Ω? over the family of weakly connected graphs with in/out-degree bounded above by d. We will first focus on the family $\mathcal{F}_d$ of *strongly connected* graphs with in/out-degree bounded above by d. A simple transformation explained in Sec. 6.10 allows to extend the result to weakly connected graphs with in/out-degree bounded by d.

The main design difficulty comes from the fact that the information given by the oracle does not allow to distinguish between the presence of a single or more leaders. Thus, a leader should try to kill possible other leaders, when avoiding a scenario where all leaders are killed infinitely often. This metaphor comes from [46] – leaders sending *bullets* for killing other leaders, and may protect themselves with *shields*. Although the protocol in [46] is not simple, the ring topology is of great help. For arbitrary graphs, managing bullets and shields is much more complicated, and agents must in some sense keep a trace of them. As the agents are finite-state, a bounded degree is needed for implementing such a management.

As a basic tool for our protocol, we use the 2-hop coloring self-stabilizing population protocol, denoted by $2HC$, proposed in [10]. A 2-hop coloring is a coloring such that all neighbours of the same agent have distinct colors. We denote by $Colors$ the corresponding set of possible colors. The protocol $2HC$ uses a set $Colors$ of size $O(d^2)$.

The Protocol $\mathcal{A}_d$

The input variables (read-only) of $\mathcal{A}_d$ at each agent x are: the *oracle output* $\Omega?_x$ (values in $\{0,1\}$); and the *agent color* c_x (values in $Colors$), which stores the output of $2HC$. The working variables are: the *leader bit* $leader_x$ (values $\{0,1\}$); the *bullet vector* $bullet_x$ (vector with values in $\{0,1\}$ indexed by $Colors$); and the *shield vector* $shield_x$ (vector with values in $\{0,1\}$ indexed by $Colors$).

The idea of the protocol is the following. An agent may hold several shields (resp. bullets), each of them waiting to be forwarded to an out-neighbour, from initiator to responder, with associated color, lines 14 – 18 (resp. in-neighbour, from responder to initiator, lines 7 – 12). The information required for implementing this is encoded in the shield and bullet vectors. The purpose of the bullets is to kill leaders (line 10), whereas the purpose of the shields is to protect them by absorbing bullets (line 17). A leader is created when the oracle reports that there are no leaders in the system (lines 2, 3). When a leader is created, it comes with (loads) a shield for every color (line 5), and thus is protected from any bullet that could come from one of its out-neighbors. To maintain the protection, each time an agent receives a shield from its in-neighbor, it reloads shields for every color (line 16). Dually, any time an agent receives a bullet, it reloads bullets for every color (line 11). In addition, whenever a leader interacts as an initiator, it loads bullets for every color (line 22).

Algorithm 8: Protocol $\mathcal{A}_d$ - initiator x, responder y

```
1  (Create a leader at x, if needed);
2  if Ω?_x = 0 then
3      leader_x ← 1;
4      ∀c ∈ Colors, bullet_x[c] ← 1;
5      ∀c ∈ Colors, shield_x[c] ← 1;
6  end
7  (Move bullet from y to x, if any);
8  if bullet_y[c_x] = 1 then
9      if shield_x[c_y] = 0 then
10         leader_x ← 0;
11         ∀c ∈ Colors, bullet_x[c] ← 1;
12     bullet_y[c_x] ← 0;
13 end
14 (Move shield from x to y, if any);
15 if shield_x[c_y] = 1 then
16     ∀c ∈ Colors, shield_y[c] ← 1;
17     bullet_y[c_x] ← 0;
18     shield_x[c_y] ← 0;
19 end
20 (Load bullets if x is a leader);
21 if leader_x = 1 then
22     ∀c ∈ Colors, bullet_x[c] ← 1;
```

Correctness

Consider a strongly connected graph G of degree (in and out degree together) less than or equal to d. For the sake of clarity, in any execution we consider, we assume that the protocol $2HC$ permanently outputs a correct 2-hop coloring from the beginning (variables c_x, for every agent x).

A *path* in G is a sequence of agents $\pi = x_0 \dots x_r$ such that (x_i, x_{i+1}) is a directed edge of G. If $x_0 = x_r$, π is a *loop at* x_0. If u is an agent that appears in π, we denote it by $u \in \pi$, and by $ind_\pi(u)$ the index of the first occurrence of u in π, i.e. the minimum i such that $x_i = u$. If (x, y) is an edge of G, we say that x *has a shield against* y if $shield_x[c_y] = 1$. Similarly, we say that y *has a bullet against* x if $bullet_y[c_x] = 1$.

Definition 7 (Protected Leader)**.** *Consider a loop $\pi = x_0 \dots x_{r+1}$ at a leader λ $(= x_0 = x_{r+1})$. We say that λ is a* leader protected in π *if there exists $i \in \{0, \dots, r\}$ such that x_i has a shield against x_{i+1} and, if $i \geq 1$, x_i is not a leader and has no bullet against x_{i-1}. In addition, for every $j \in \{1, \dots, i-1\}$, x_j is not a leader, has no shield against x_{j+1} and no bullet against x_{j-1}. The agent x_i is the* protector of λ in π*; the path $x_0 \dots x_i$ is the* protected zone in π*. The agent λ is a* protected leader *if it is protected in every loop at λ.*

Note that a new leader or a leader that receives a shield becomes protected by loading shields for every color.

As before, given an execution E, E_∞ denotes the maximum (infinite) suffix of E such that each couple (γ, α) (γ being a configuration, and α an input assignment) in E_∞ occurs infinitely often. IR_E denotes the (finite) set of configurations occurring in E_∞, i.e., the set of "infinitely recurrent" configurations in E.

Lemma 11. *If $\gamma \in IR_E$ has a protected leader, then every configuration in IR_E has a protected leader.*

Proof. Consider a couple (γ, α) that occurs in E_∞, γ being a configuration (in IR_E) and α an input assignment. Let γ' be a configuration s.t. (γ, α) goes to γ' via an action involving a directed edge (x, y). By global fairness, $\gamma' \in IR_E$ too, and we show that it has a protected leader.

Note that when a leader is created, it is already protected by itself since it has a shield against every of its out-neighbors (line 5). We thus focus on

transition rules that do not involve the creation of a leader. Let λ be a protected leader in γ and π be any loop at λ. Let μ be the protector of λ in π. If x and y do not appear in the protected zone in π, then after the transition, the states of the agents in the protected zone have not changed and λ is still protected in π. Then, assume that x or y appear in the protected zone and let $z \in \{x, y\}$ be the agent with lowest index $ind_\pi(z)$. By the choice of z, $ind_\pi(z) \leq ind_\pi(\mu)$.

Consider first the case $ind_\pi(z) < ind_\pi(\mu)$. If $z = x$, then z cannot receive a bullet (from y), i.e., either x has a shield against y or y has no bullets against x. Otherwise, the path that goes from λ to (the first occurrence of) $z = x$ followed by any path that goes from y to λ yields a loop within which λ is not protected in γ; hence a contradiction. Hence, if $z = x$, after the transition, λ is still protected by μ in π. Now, if $z = y$, y may only receive a shield, and thus, after the transition, λ is still protected in π (by μ or y).

Now, assume that $ind_\pi(z) = ind_\pi(\mu)$. This implies that $z = \mu \in \{x, y\}$, and that every agent in the protected zone, except μ, is different from x and y. If $\mu = y$, then during the transition, μ may only receive a shield (which merges with its own shield); hence, λ is still protected by μ in π after the transition. The case $\mu = x$ is more complicated. First consider the subcase where y is not the agent that follows the first occurrence of μ in π. Then μ cannot receive a bullet during the transition, otherwise, the same argument as above shows the existence of a loop at λ within which λ is not protected in γ. After the transition, (the first occurrence of) μ has still a shield against the agent right after it, which proves that λ is still protected in π. Consider now the subcase where y is the agent that follows the first occurrence of μ in π. If y is not a leader, then after the transition, y becomes the new protector of λ in π. If y is a leader, then after the transition, λ is no longer protected, but y is protected since the reception of a shield produces shields for every color. In both cases, after the transition, there is a protected leader in γ'.

We thus have shown that, in all cases, γ' contains a protected leader. Given any configuration $\gamma'' \in IR_E$, there must be a sequence of actions from (γ, α) to (γ'', α'') during E_∞, for some input assignment α''. Since γ has a protected leader, the proof shows that every configuration in this sequence, and in particular γ'', has also a protected leader. Therefore, any configuration γ'' in IR_E has a protected leader. □

Lemma 12. *If no configuration in IR_E has a leader, then in every input assignment in E_∞, $\Omega?_x = 0$ for some agent x. If every configuration in IR_E has a leader, then in every input assignment in E_∞, $\Omega?_x = 1$ for every agent x.*

Proof. This stems from the definition of Ω?. □

Lemma 13. *Every configuration in IR_E has a leader.*

Proof. Assume that some configuration γ in IR_E lacks a leader. On the one hand, if no configuration in IR_E has a leader, then by Lemma 12, in every input assignment in E_∞, $\Omega?_x = 0$ for some agent x. Hence, in E_∞, during a transition involving x, a *protected* leader is created (lines 2 – 5). On the other hand, if IR_E contains a configuration γ' with a leader, then there is a sequence of actions from (γ, α) to (γ', α') for some input assignments α, α', since both γ and γ' occur infinitely often in E_∞. According to the protocol, during some

of these actions, a *protected* leader must be created. In both cases, there is a configuration $\gamma'' \in IR_E$ with a protected leader. By Lemma 11, this implies that every configurations in IR_E, and in particular γ, has a protected leader; hence a contradiction. □

Lemma 14. *All configurations in IR_E have the same number of leaders.*

Proof. By the lemmas 12 and 13, in every input assignment in E_∞, $\Omega?_x = 1$ for every agent x. Thus no leader is created during E_∞. Assume that there exists two configurations γ, γ' in IR_E such that the number l of leaders in γ is different from the number l' of leaders in γ'. Without loss of generality, we can assume $l < l'$. By definition of E_∞, there must be a sequence of actions from (γ, α) to (γ', α') for some input assignments α, α'. The fact that $l < l'$ implies that during this sequence a leader is created; hence a contradiction. □

Lemma 15. *No configuration in IR_E contains an unprotected leader.*

Proof. By the lemmas 12 and 13, in every input assignment in E_∞, $\Omega?_x = 1$ for every agent x. Assume that $\gamma \in IR_E$ contains an unprotected leader λ. Since λ is not protected in γ, there exists a path $\pi = x_0 \dots x_r$ from $x_0 = \lambda$ to some agent x_r such that for every $0 \leq i < r$, x_i has no shield against x_{i+1}, and x_r is either a leader or has a bullet against x_{r-1}. If x_r is a leader, then in any transition where it is the initiator, it creates a bullet against x_{r-1}. Thus, in both cases, there is a bullet that, by moving (backward) along this path to λ, can kill this non-protected leader. Thus, a configuration γ' within which λ is not a leader is reachable from γ. During the sequence of actions from γ to γ', no leaders are created. Thus, γ' has fewer leaders than γ. The global fairness ensures that $\gamma' \in IR_E$. This contradicts Lemma 14. □

Proposition 20. *The protocol $\mathcal{A}_d$ solves the problem $\mathcal{ELE}$ using $\Omega?$ (i.e., $\Omega? \succcurlyeq \mathcal{ELE}$) over strongly connected graphs with degree less than or equal to d.*

Proof. By the previous lemmas, every configuration $\gamma \in IR_E$ has $l \geq 1$ protected leaders and no unprotected leaders; and in every input assignment in E_∞, $\Omega?_x = 1$ for every agent x. Assume, by contradiction, that $l \geq 2$. Let λ_1, λ_2 be two protected leaders in γ. Consider a shortest path p_1 (resp. p_2) from λ_1 to λ_2 (resp. from λ_2 to λ_1). Consider the loop $\pi_1 = p_1 p_2$ at λ_1, and the loop $\pi_2 = p_2 p_1$ at λ_2. Denote by μ_1 (resp. μ_2) the protector λ_1 (resp. λ_2) in π_1 (resp. π_2). By construction, in γ, the first occurrence of μ_1 (resp. μ_2) is in p_1 (resp. p_2). By definition and according to the protocol, it is possible to move the (first occurrence of the) protector μ_1 to the position right before λ_2. Another movement makes the protector transfer its shield to λ_2, thus turning λ_1 into a non-protected leader (λ_2 is still a protected leader). Then λ_2 can fire a bullet that kills λ_1. Since no leader is created during the described sequence of actions ($\Omega?_x = 1$ for every agent x), the reached configuration γ' has $l - 1$ leaders. As global fairness ensures that $\gamma' \in IR_E$, this contradicts Lemma 14. Therefore, all configurations in IR_E have a unique leader. Since a leader cannot move, there is a permanent leader. □

6.7 SSLE with Ω?(2) over Arbitrary Graphs

In this section, we show that $\mathcal{ELE}$ can be implemented using Ω?(2) over the family of arbitrary (connected) graphs. According to Sec. 6.10, it is sufficient to prove the result over strongly connected graphs. The idea of the protocol is simple. A leader moves when it "knows" there are other leaders and does not move when it "knows" it is the unique leader, this information being provided by the oracle. We define the protocol as follows. The input alphabet is $\{0, 1, 2\}$, the state space is $\{\bullet, \circ\}$ where $\bullet$ (resp. $\circ$) stands for leader (resp. non leader). The rules are :

$$\circ, \circ \xrightarrow{0,0} \bullet, \circ \tag{6.1}$$

$$\bullet, \circ \xrightarrow{2,2} \circ, \bullet \tag{6.2}$$

$$\bullet, \bullet \xrightarrow{*,*} \bullet, \circ \tag{6.3}$$

The symbol $*$ means "any possible value". During a transition, the output values are the state values; they are omitted in the rules above. In every case not listed above, the states are unchanged. Basically, a leader is created whenever the oracle outputs 0 (rule 6.1). The leaders keep moving in the graph while the oracle outputs 2 (rule 6.2). When two leaders meet, one of them disappears (rule 6.3).

Proposition 21. *The protocol above is a self-stabilizing implementation of $\mathcal{ELE}$ using Ω?(2) over strongly connected graphs.*

Proof. Consider a strongly connected graph G and consider a globally fair execution E of the protocol. Assume that every configuration γ in IR_E lacks a leader. The definition of Ω?(2) implies that every input assignment that occurs in E_∞ assigns 0 to every agent. But, by rule (1), γ can reach a configuration γ' with a leader, and the global fairness ensures that $\gamma' \in IR_E$; whence a contradiction. Thus, there exists a configuration $\gamma \in IR_E$ that has a leader. The rule (3) (the only rule to kill a leader) implies that, for any input assignment α and for any configuration γ' such that $(\gamma, \alpha) \to \gamma'$, γ' has a leader. Now, consider any $\gamma'' \in IR_E$. By definition of E_∞, there must be a sequence of steps from (γ, α) to (γ'', α'') during E_∞, and the previous argument shows that every configuration during this sequence has a leader; in particular γ''.

Thus, every configuration in IR_E has at least one leader. The definition of Ω?(2) implies that any input assignment in E_∞ does not assign 0 to any agent. Therefore, no leaders are created during E_∞. If there were two configurations in IR_E with different number of leaders, then there would be a step in E_∞ during which a leader is created; this is impossible. Hence, every configuration in IR_E has the same number c of leaders. If $c \geq 2$, then the definition of the oracle implies that every input assignment in E_∞ assigns 2 to every one. Since the graph is strongly connected, from any configuration $\gamma \in IR_E$ with $c \geq 2$ leaders, it is possible (via rule (2)) to move the two leaders to two neighbor nodes and to kill one of them (via rule (3)), thus reaching a configuration $\gamma' \in IR_E$ with less than c leaders; whence a contradiction. Hence, $c = 1$, i.e. there is a unique leader in every configuration in IR_E. Then the definition of the oracle implies that every input assignment assigns 1 everywhere. Thus, during E_∞, the three rules of the protocol are disabled, and the unique leader is permanently located at some node. □

6.8 SSLE with $\Omega? \otimes \Omega?$ over Arbitrary Graphs

In Sec. 6.7, the oracle $\Omega?(2)$ seems a bit strong (the protocol is very simple). In this section, we focus on using only the oracle $\Omega?$ of Fischer and Jiang. Precisely, we exhibit a self-stabilizing solution to $\mathcal{ELE}$ using $\Omega? \otimes \Omega?$, i.e., two copies of the Fischer and Jiang's oracle, over the family of arbitrary graphs.

The Protocol $\mathcal{B}$

Alg. 9 below, referred to as the protocol $\mathcal{B}$, is a self-stabilizing solution to $\mathcal{ELE}$ using $\Omega? \otimes \Omega?$ over the arbitrary graphs.

In this protocol, each agent can be a leader or not, and a leader can be either black or white. An agent can also hold a token, and a token can be either black or white. We denote by $\Omega?^l$, resp. $\Omega?^t$, the copy of the oracle $\Omega?$ used to detect the absence of leaders, resp. tokens. As explained in Sec. 6.10, we only consider *strongly* connected graphs.

Whenever the oracle $\Omega?^l$, resp. $\Omega?^t$, outputs 0, a black leader, resp. a black token, is created. The tokens keep moving through the network by swapping between two agents during an interaction. When a black token interacts with a white leader, the leader becomes a non-leader. When a white token interacts with a black leader, the leader becomes white. When a token interacts with a leader having the same color, then both the token and the leader turn into the opposite color.

Algorithm 9: The Protocol $\mathcal{B}$

variables *agent* x
  $\Omega?^l_x$: input (read-only) from the leader detector;
  $\Omega?^t_x$: input (read-only) from the token detector;
  $leader_x$: $\bot$ (non-leader), *white* or *black*;
  $token_x$: $\bot$ (no token), *white* or *black*;
protocol *(initiator x, responder y)*
  if $\Omega?^l_x = 0$ **then** $leader_x \leftarrow black$;
  ;
  if $\Omega?^t_x = 0$ **then** $token_x \leftarrow black$;
  ;
  if $token_x = black \wedge leader_y = white$ **then** $leader_y \leftarrow \bot$;
  ;
  if $token_x = white \wedge leader_y = black$ **then** $leader_y \leftarrow white$;
  ;
  if $token_x = leader_y = black$ **then** $token_x \leftarrow leader_y \leftarrow white$;
  ;
  if $token_x = leader_y = white$ **then** $token_x \leftarrow leader_y \leftarrow black$;
  ;
  if $token_x \neq \bot \wedge token_y \neq \bot$ **then** $token_x \leftarrow \bot$;
  ;
  $token_x \leftrightarrow token_y$;

Correctness

Given an input assignment α for the Alg. 9, we denote by $\alpha.\Omega?^l_x$ (resp. $\alpha.\Omega?^t_x$) the value assigned by α to the (read-only) variable $\Omega?^l_x$ (resp. $\Omega?^t_x$). Similarly, given a configuration γ, for every agent x, we denote by $\gamma.leader_x$ (resp.

$\gamma.token_x$) the value of the variable $leader_x$ (resp. $token_x$) in the configuration γ.

Given a configuration γ, let $t(\gamma)$ (resp. $l(\gamma)$) be the total number of tokens (resp. leaders) in γ. In γ, if an agent x is a leader and an agent y holds a token (x and y not necessarily neighbours), we say that the leader at x and the token at y are *synchronized* if they have the same color. Then, we say that the configuration γ *contains a synchronized pair of leader and token.*

As before, given an execution E, E_∞ denotes the maximum (infinite) suffix of E such that each couple (γ, α) (γ being a configuration, and α an input assignment) in E_∞ occurs infinitely often. IR_E denotes the (finite) set of configurations occurring in E_∞, i.e., the set of "infinitely recurrent" configurations in E.

Lemma 16. *For every (γ, α) in E_∞, there is a unique token in γ and α assigns 1 to every variable $\Omega?^t_x$, i.e. $t(\gamma) = 1$ and $\forall x, \alpha.\Omega?^t_x = 1$.*

Proof. Assume first that for every (γ, α) in E_∞, $t(\gamma) = 0$. Then by the definition of $\Omega?^t$, for every (γ, α) in E_∞, $\alpha.\Omega?^t_x = 0$ for every agent x. By line 9, a token is created at some point during E_∞; whence a contradiction. Hence, there exists (γ', α') in E_∞ such that $t(\gamma') \geq 1$. Since the only way to reduce the number of tokens is by merging two existing tokens (line 19), for every configuration γ such that $(\gamma', \alpha') \to \gamma$, $t(\gamma) \geq 1$. Hence, for every couple (γ, α) in E_∞, $t(\gamma) \geq 1$. The definition of $\Omega?^t$ involves that for every (γ, α) in E_∞, $\alpha.\Omega?^t_x = 1$ for every agent x. This disables the creation of token during E_∞. Thus, the number of tokens cannot increase during E_∞. Actually, since each couple (γ, α) occurs infinitely often in E_∞, the number of tokens during E_∞ is constant, say t_0. The previous argument shows that $t_0 \geq 1$. Assume that $t_0 \geq 2$. Then, by global fairness, there is a configuration in E_∞ in which two tokens are located at two neighbouring nodes. From this configuration, there is a reachable configuration γ' resulting from the interaction of these two neighbours, such that $t(\gamma') \leq t_0 - 1$. The global fairness ensures that γ' is in E_∞; whence a contradiction. Hence, $t_0 = 1$, i.e., there is a unique token during E_∞. □

Lemma 17. *Consider a configuration γ that contains a synchronized pair of leader and token such that $l(\gamma) \geq t(\gamma) = 1$. Consider an input assignment α that assigns 1 to every variable $\Omega?^t_x$, i.e., for all x, $\alpha.\Omega?^t_x = 1$. Then for any configuration γ' such that $(\gamma, \alpha) \to \gamma'$, γ' contains a synchronized pair of leader and token and $l(\gamma') \geq t(\gamma') = 1$.*

Proof. In Alg. 9, if line 7 is executed, then the number of leader increases. Line 9 is not executed since $\alpha.\Omega?^t_x = 1$ for every x.

If line 11 is executed, then $l(\gamma') = l(\gamma) - 1$ and $t(\gamma') = t(\gamma) = 1$. Since γ contains a synchronized pair of leader and token and since the unique token is black in γ, there must be a black leader in γ (not involved in the interaction). Thus $l(\gamma) \geq 2$, $l(\gamma') \geq t(\gamma') = 1$ and γ' also contains a synchronized pair of leader and token.

If line 13 is executed, then $l(\gamma') = l(\gamma)$ and $t(\gamma') = t(\gamma) = 1$, whence $l(\gamma') \geq t(\gamma') = 1$. Since γ contains a synchronized pair of leader and token and since the unique token is white in γ, there must be a white leader in γ

(not involved in the interaction). Hence, γ' also contains a synchronized pair of leader and token.

If line 15 is executed, then $l(\gamma') = l(\gamma)$ and $t(\gamma') = t(\gamma) = 1$, whence $l(\gamma') \geq t(\gamma') = 1$. The interaction involves a synchronized pair of leader and token, and since both the leader and the token flip their color, γ' also contains the same synchronized pair of leader and token. The same argument applies for line 17.

Finally, line 19 cannot be executed since $t(\gamma) = 1$, and line 21 just swap the token values. Therefore, in all cases, γ' contains a synchronized pair of leader and token and $l(\gamma') \geq t(\gamma') = 1$. □

Lemma 18. *There exists a configuration γ in E_∞ that contains a synchronized pair of leader and token such that $l(\gamma) \geq t(\gamma) = 1$.*

Proof. We prove the result by contradiction. By Lem. 16, we already know that every configuration in E_∞ contains a unique token. Hence, assume that, for every configuration γ in E_∞, any leader in γ (if any) does not have the same color as the (unique) token in γ. Note that, if every configuration γ in E_∞ has no leader, then the definition of $\Omega?^l$, the global fairness and the rules of the protocol involve that a (black) leader is created at some point in E_∞; whence a contradiction. Hence, there exists a configuration γ in E_∞ which has at least one leader, $l(\gamma) \geq t(\gamma) = 1$.

By our hypothesis, every leader in γ has the same color, opposite to the color of the token. Consider the case where the token is white. Thus all the leaders in γ are black. Whatever the sequence of input assignment is, it is possible to reach from γ a configuration γ' with one white leader and one white token, simply by moving the white token towards one of the black leaders, and apply the rule of the protocol that turns this leader white. The configuration γ' has a synchronized pair of leader and token, and $l(\gamma') \geq t(\gamma') = 1$. By the global fairness, γ' must belong to E_∞; whence a contradiction.

Consider the case where where the token is black. Thus all the leaders in γ are white. By moving the token, it is possible to turn all the leaders into non-leaders. Hence, there exists a configuration γ' occurring in E_∞ with no leaders and one black token. Now since γ occurs in E_∞, it occurs infinitely many times in E_∞, and there is a sequence of steps $(\gamma', \alpha') \dots (\gamma, \alpha)$ in E_∞. During this sequence, a leader is created. Before this creation, the unique token stays black since it interacts with no leader. The rules of the protocol involve that the first created leader is black. Hence, there exists a configuration γ'' in E_∞ which contains a synchronized pair of leader and token, and such that $l(\gamma'') \geq t(\gamma'') = 1$; whence a contradiction. □

Lemma 19. *For every (γ, α) in E_∞, γ contains a synchronized pair of leader and token, $l(\gamma) \geq t(\gamma) = 1$ and for every agent x, $\alpha.\Omega?^l_x = \alpha.\Omega?^t_x = 1$.*

Proof. By Lem. 16, we already know that for every (γ, α) in E_∞, $t(\gamma) = 1$ and for every agent x, $\alpha.\Omega?^t_x = 1$. Also by Lem. 18, we know that there exists a (γ, α) in E_∞, such that γ contains a synchronized pair of leader and token, and $l(\gamma) \geq t(\gamma) = 1$. These two results, and Lem. 17 ensure that every (γ, α) in E_∞ contains a synchronized pair of leader and token, and $l(\gamma) \geq t(\gamma) = 1$. Then, the definition of $\Omega?^l$ involves that every input assignment α occurring in E_∞ is such that for all x, $\alpha.\Omega?^l_x = 1$. □

Proposition 22. *Alg. 9 is a self-stabilizing implementation of $\mathcal{ELE}$ using $\Omega?\otimes\Omega?$. Precisely, in any execution, there exists exactly one agent λ such that for every configuration γ in E_∞, $\gamma.leader_\lambda \neq \bot$ and for every agent $\mu \neq \lambda$, $\gamma.leader_\mu = \bot$.*

Proof. By Lem. 19, we know that during E_∞, the leader detector $\Omega?^l$ outputs 1 everywhere. Hence, no leader is ever created during E_∞. This involves that the number of leaders (greater than or equal to 1) cannot increase during E_∞. Actually, since each (γ, α) in E_∞ occurs infinitely often in E_∞, the number of leaders is constant during E_∞. We denote by c this constant; we already know that $c \geq 1$.

Assume that $c \geq 2$. Consider a configuration γ occurring in E_∞. We know that γ contains a synchronized pair of leader and token and that $l(\gamma) = c \geq 2$, $t(\gamma) = 1$. We now describe scenarios that produce a configuration γ' out of γ, such that γ' contains a unique leader (synchronized with the unique token).

case (a). The unique token in γ is black. There must be a black leader since γ contains a synchronized pair of leader and token. By global fairness, it is possible to move the token near this leader, and to turn them both white. Then we come down to case (b).

case (b). The unique token in γ is white. By moving the token to meet every black leaders, we can turn all the black leaders white. Then by global fairness, we can assume that there are no black leaders in γ. Still by global fairness, the following sequence of moves is possible. First, the white token meets a white leader and they both turn black. Then the black token successively meets the white leaders and turn them into non-leaders. The resulting configuration has a unique (black) leader (and a unique black token). The global fairness ensures that this configuration occurs in E_∞; whence a contradiction with the fact that the number of leaders is $c \geq 2$.

Therefore, $c = 1$, i.e., there is a unique leader in every configuration during E_∞. Since every configuration in E_∞ contains a synchronized pair of leader and token, in each configuration, the unique leader must be synchronized with the unique token. Since a leader cannot be turned into a non-leader by a token with which it is synchronized, the unique leader is the same for every configuration in E_∞. Precisely, there exists an agent λ such that for every configuration γ in E_∞, $\gamma.leader_\lambda \neq \bot$ and for every agent $\mu \neq \lambda$, $\gamma.leader_\mu = \bot$. □

6.9 Ω? is not stronger than $\mathcal{ELE}^{\otimes k}$ over a Non-Simple Graph Family

We show that there is no self-stabilizing implementation of Ω? using $\mathcal{ELE}^{\otimes k}$ (i.e. k parallel instances of $\mathcal{ELE}$) for any $k \geq 1$, over a non-simple family $\mathcal{F}$ of graphs. Recall that a family $\mathcal{F}$ is non-simple if there are graphs $G, G_1, G_2 \in \mathcal{F}$ such that G_1, G_2 are disjoint subgraphs of G.

Proposition 23. *For any non-simple family of graphs $\mathcal{F}$, there is no self-stabilizing population protocol $\mathcal{A}$ implementing Ω? over $\mathcal{F}$ using the behaviour $\mathcal{ELE}^{\otimes k}$ ($k \geq 1$). In other words, there is no composition $B = \mathcal{ELE}^{\otimes k} \circ Beh(\mathcal{A}) \subseteq \Omega?$.*

Proof. We prove the result by contradiction using a classical partitioning argument. Assume such a protocol $\mathcal{A}$ and consider a graph $G \in \mathcal{F}$, such that there are two disjoint subgraphs of G, G_1 and G_2 that are also in $\mathcal{F}$. Without loss of generality, we assume the output of the protocol are encoded in the states.

Every execution E of $\mathcal{A}$ has an input trace (T, T_{in}), where T is an output trace of $\mathcal{ELE}^{\otimes k}$ and T_{in} represents the input trace of $\Omega?$. The trace T has values in $\{0,1\}^k$. We choose T to be the constant trace which outputs $(1,\dots,1)$ at some agent $\lambda \in G_1$, and $(0,\dots,0)$ everywhere else. We denote by β this specific assignment ($T = \beta\beta\dots$). On the other hand, we choose the trace T_{in} (with values in $\{0,1\}$) to be the constant trace which outputs 1 at some agent $\mu \in G_2$. We denote by α this specific assignment ($T_{in} = \alpha\alpha\dots$).

The choice of T and T_{in} yields an execution E, say with schedule S, and an output trace T_{out}. Since the composition B is a sub-behaviour of $\Omega?$, the output trace T_{out} belongs to $\Omega?(G, S, T_{in})$. Hence, (Δ) the output trace T_{out} has a suffix equal to the constant trace assigning 1 to every agent. Since the outputs are encoded in the states of the agents, it means that, for every couple $(\gamma, (\beta, \alpha))$ in E_∞, the output associated to γ assigns 1 to every agent.

If we restrict $(\gamma, (\beta, \alpha))$ to the graph G_1, we obtain a configuration and input assignment $(\gamma^1, (\beta^1, \alpha^1))$. The agent λ is still the unique agent to be assigned $(1,\dots,1)$ by β^1, and α^1 assigns 0 to every agents in G_1. Since the protocol is self-stabilizing, and since $G_1 \in \mathcal{F}$, there is a sequence of actions, involving all the agents of G_1 and having the constant trace with the assignment (β^1, α^1) during the sequence. This leads to a configuration γ'^1 that outputs 0 at at least one agent in G_1. This involves that there is a finite execution $(\gamma, (\beta, \alpha))(\gamma_1, (\beta, \alpha))(\gamma_2, (\beta, \alpha))\dots(\gamma', (\beta, \alpha))$ such that γ' outputs 1 at the agents of G_2 and 0 at some agent in G_1. The global fairness ensures that γ' occurs in E_∞. This implies that T_{out} outputs 0 at some agent (in G_1) infinitely often. This contradicts (Δ). □

6.10 From Strongly to Weakly Connected Graphs

In this section, we show how to extend the results on strongly connected graphs to the weakly connected graphs. Given a weakly connected graph G, the *symmetric closure* G_{sym} of G is the graph with the same set of vertices, $Vert(G_{sym})$ such that a couple $(x, y) \in Edges(G_{sym})$ if and only if $(x, y) \in Edges(G)$ or $(y, x) \in Edges(G)$. It is straightforward to check that G_{sym} is strongly connected.

Proposition 24. *Let $\mathcal{F}$ be a family of strongly connected graphs, and $\mathcal{WF}$ be the family of (weakly connected) graphs G whose symmetric closure G_{sym} belongs to $\mathcal{F}$.*

Given any population protocol $\mathcal{A}$ implementing the behaviour $\mathcal{ELE}$ over the family $\mathcal{F}$, there is a population protocol $\mathcal{A}'$ (given in the proof) implementing $\mathcal{ELE}$ over the family $\mathcal{WF}$

Proof. We give a constructive proof. Without loss of generality, we assume that the outputs of $\mathcal{A}$ are encoded in the states of the agents. We show how to transform $\mathcal{A}$ into a population protocol $\mathcal{A}'$. Given $\mathcal{A}$, we define below a (possibly) non-deterministic protocol $\mathcal{A}^{ND}$. It can be transformed into a de-

terministic one by the transformer proposed in [10] (since $\mathcal{ELE}$ is an elastic behaviour).

$\mathcal{A}^{ND}$ has the same state space, inputs and outputs as $\mathcal{A}$, and the following transition rules.

$$p, q \xrightarrow[o_1,o_2]{i_1,i_2} p', q' \text{ in } \mathcal{A}^{ND} \Leftrightarrow \begin{cases} p, q \xrightarrow[o_1,o_2]{i_1,i_2} p', q' \\ \text{or } q, p \xrightarrow[o_2,o_1]{i_2,i_1} q', p' \end{cases} \quad \text{in } \mathcal{A} \tag{6.4}$$

Note that, if $\mathcal{A}$ is a symmetric deterministic protocol, then $\mathcal{A}^{ND}$ is deterministic.

Intuitively, $\mathcal{A}^{ND}$, executing over a weakly connected graph G, simulates $\mathcal{A}$ over a strongly connected graph which is the symmetric closure G_{sym} of G. Alternatively, it is as if $\mathcal{A}^{ND}$ simulated a scheduler, over a non directed graph induced by G, which could choose at every interaction which agent is the initiator, and which is the responder.

We now show that $\mathcal{A}^{ND}$ also implements $\mathcal{ELE}$ as $\mathcal{A}$ over a family of weakly connected graphs. Consider a (globally fair) execution E of $\mathcal{A}^{ND}$ on G

$$\dots \gamma_t \xrightarrow{\sigma_t} \gamma_{t+1} \dots \tag{6.5}$$

where σ_t is the action triggered at time t. We build a globally fair execution E' of $\mathcal{A}$ on G_{sym} with the same sequence of configurations. Indeed, for any t, if

$$\sigma_t = \left((x, y),\ p, q \xrightarrow[o_x,o_y]{i_x,i_y} p', q' \right) \tag{6.6}$$

then we define

$$\sigma'_t = \begin{cases} \sigma_t \text{ if } p, q \xrightarrow[o_x,o_y]{i_x,i_y} p', q' \text{ is a rule of } \mathcal{A} \\ \left((y, x),\ q, p \xrightarrow[o_y,o_x]{i_y,i_x} q', p' \right) \text{ otherwise} \end{cases} \tag{6.7}$$

Is not difficult to check that the following

$$\dots \gamma_t \xrightarrow{\sigma'_t} \gamma_{t+1} \dots \tag{6.8}$$

is a globally fair execution of $\mathcal{A}$ on G_{sym} (with the same sequence of configurations). Hence, since $\mathcal{A}$ solves $\mathcal{ELE}$ on G_{sym}, the protocol $\mathcal{A}^{ND}$ solves $\mathcal{ELE}$ on G. □

Remark 2. *Note that we have only used the fact that electing a leader on G_{sym} is similar to electing a leader on G. Hence, the given proof also applies to any behaviour B (other than $\mathcal{ELE}$) with a similar relation between the legal histories on G and the legal histories on G_{sym}.*

Part II

State-Machine Replication

Chapter 7

Introduction

7.1 Introduction

State-Machine Replication

Imagine a system processing requests from clients and replying adequate responses. If this system is implemented on a unique machine, then the requests are likely to be processed slowly, and, first and foremost, the whole system is broken whenever the machine fails. A very common approach to provide a reliable system is to replicate the program (state-machine) over many servers (replicas). The basic idea is that if some of the replicas fail, then the system should be able to keep processing the requests.

However, one does not want the system to give absurd responses to the clients requests. For instance, if there are three copies of the same book left in the bookshop's storage, then we do not want the system to sell the book to more than three different clients. Indeed, the system must process the requests in a *coherent* way. But what is coherence ? From a very general perspective, it is natural to require that the whole system behaves globally as a unique state-machine processing the different requests sequentially. In other words, the system is required to be *linearizable* [50].

This is a difficult issue because, in a distributed system, the requests may not arrive at different replicas in the same order. Hence, the replicas must somehow *agree* on the order of requests when executing them. One approach consists in relating this issue to the consensus problem. Indeed, if all the replicas initially share the same state and if they execute the same requests in the same order, then the system is coherent from the client's point of view. It is possible then to picture the system as a sequence of consensus instances that decide on the request to execute at each step. Roughly speaking, the requirements are the following: *(safety)* two processes cannot decide on different requests for the same step; *(liveness)* every process eventually decides on a request for every step, unless it crashes.

However, in an asynchronous message-passing system prone to crash failures, solving a single consensus instance has been proven impossible [47]. This hinders the possibility of a state-machine replication protocol. Yet, Lamport has provided an algorithmic scheme, namely Paxos [56, 57], that partially satisfies the requirements of state-machine replication in the following sense. The

safety property is always guaranteed. But, the liveness property requires additional assumptions; usually any means to elect a unique leader for a long enough period of time.

Note that the original formulation [57] presented Paxos as a (partial) solution to the consensus problem, but its actual purpose is to implement a replicated state-machine. Since then, many improvements have been proposed, e.g., Fast Paxos [59], Generalized Paxos [58], Byzantine Paxos [60], and the study of Paxos has become a subject of research on its own. The extreme usefulness of such an approach is proven daily by the usage of this technique by the very leading companies [30].

Practical Self-Stabilization

Unfortunately, none of these approaches deal with the issue of transient faults, i.e., punctual corruptions of the data that may put the system in an arbitrary configuration. In the context of replicated state-machines, these faults may induce two kinds of effect. First, they can corrupt the local states of the replica, and thus, even if the replicas execute the same requests in the same order, they will permanently give wrong answers to the clients: the linearizability of the system is altered. However, this is not the worse issue. Indeed, if the replicas still have the possibility to agree on something, then they can also agree on a common state to start with. A much worrying issue is when the transient fault corrupts the core of the algorithm that synchronizes the replicas. For instance, the replicas may be permanently unable to process new requests, or, they execute different sequences of requests. This last issue threatens both the linearizability and the liveness of the system.

Self-stabilization was introduced in the seminal paper [39] of Dijkstra. Roughly speaking, a self-stabilizing system is able to recover from any transient fault after a finite period of time. In other words, after the last transient fault, a self-stabilizing system ensures that eventually the processors behave according to the specifications of the problem. Since the effect of a transient fault is to put the system in an arbitrary configuration, and since we only focus on the suffix after the last transient fault, an equivalent formulation states that a self-stabilizing system, started in an arbitrary configuration, eventually behaves correctly forever.

The nature of self-stabilization implies that it only concerns "live" problems, i.e., problems in which the processors must guarantee a service forever. There is no obvious meaning to a self-stabilizing solution of a "one-shot" problem. For instance, if an algorithm claims to be a solution of the consensus problem, then transient fault may force the replicas to decide on their own input values right from the beginning. On the other hand, the closely related-problem of the replicated-state machine is a live problem. We cannot prevent, though, transient faults making the replicas decide on different requests to execute at some point in time. Yet, we look for means to guarantee that *eventually* the replicas will execute the same requests in the same order from a common state.

Completing this goal is rather difficult. Indeed, one of the main ingredients of any Paxos-based replicated state-machine algorithm is its ability to distinguish old and new messages. At a very abstract level, one uses natural numbers to timestamp data, i.e., each processor is assumed to have an infinite memory. At a more concrete level, the processes have a finite memory, and the simplest

timestamp structure is given by a natural number bounded by some constant $2^{\mathfrak{b}}$, where $\mathfrak{b}$ is the size of the register. Roughly speaking, this implies that the classic Paxos-based replicated state-machine approach is able to distinguish messages in a window of size $2^{\mathfrak{b}}$.

This constant is so large that it is sufficient for any practical purposes, as long as transient faults are not considered. For example, if a 64-bits counter is initialized to 0, incrementing the counter every nanosecond will last about 500 years before the maximum value is reached; this is far greater than any concrete system's timescale. But, a transient fault may corrupt the timestamps (e.g. counters set to the maximum value) and, thus, lead to replicas executing requests in different order or being permanently blocked although the usual liveness related conditions (e.g. unique leader) are satisfied.

This remark leads to a weaker form of self-stabilizing systems. Indeed, in the original self-stabilization formulation, one looks for a suffix of the execution (started in an arbitrary configuration) during which everything behaves correctly. We weaken this condition by requiring only that the execution (started in an arbitrary configuration) contains a finite factor, or segment, of execution during which the system behaves correctly; this segment being "long enough" compared to some predefined timescale. By a long enough segment, we mean a segment of execution whose longest causal chain of events has length greater than $2^{\mathfrak{b}}$. An algorithm satisfying this weaker self-stabilization is called a *practically self-stabilizing* algorithm.

Practical self-stabilization may look weak at first sight, but one should notice that any implementation of the original Paxos (no transient faults assumed) behaves correctly until the timestamps reach the maximum value $2^{\mathfrak{b}}$. This yields a correct but finite execution of length $O(2^{\mathfrak{b}})$, which is *practically infinite*, i.e., largely greater than any concrete system's timescale.

Our goal, in this part of the thesis, is to enhance the original Paxos algorithm so that it may start in an arbitrary configuration and still reach a point from which the system behaves correctly for a finite but practically infinite period of time. To sum up, we provide a new bounded timestamp architecture and describe the core of a practically self-stabilizing replicated state-machine (based on Paxos), in an asynchronous message passing communication environment prone to crash failures.

This work will appear in the proceedings of the Netys 2014 conference. A preliminary version has been published in [19].

7.2 Overview

In Chap. 8, we specify the model and the notations (Sec. 8.1), and we present the original Paxos algorithm to the extent we need for our purpose (Sec. 8.2). In Sec. 8.3, we informally explain how to make Paxos self-stabilizing. In Chap. 9, we formally describe our algorithm, and prove its main properties in Chap. 10.

7.3 Related work

If a process undergoes a transient fault, then one can model the process behaviour as a byzantine behaviour. In [28], Castro and Liskov present a concrete

replicated state-machine algorithm that copes with byzantine failures. Lamport presents in [60] a byzantine tolerant variant of Paxos which has some connections with Castro and Liskov's solution. Note, however, that in both cases, the number of byzantine processes must be less than the third of the total number of processes. This is related to the impossibility of a byzantine tolerant solution to consensus where more than a third of the system are byzantine. The issue of bounded timestamp system has been studied in [41] and [52], but these works do not deal with self-stabilization.

The first work, as far as we know, on a self-stabilizing timestamp system is presented in [2], but it assumes communications based on a shared memory. In [4], the authors present the notion of practical[1] stabilization, and provide an implementation of a practically self-stabilizing single-writer multi-reader atomic register. Doing so, they introduce a self-stabilizing timestamp system. However, their approach assumes that a single processor (the writer) is responsible for incrementing timestamps. Our timestamp system is a generalization which allows many processors to increment timestamps. Finally, in [44], the authors present the first practically replicated state-machine in the case of shared memory based communications.

[1] "pragmatic" in their text.

Chapter 8

Towards a Self-Stabilizing Replicated State-Machine

8.1 Model

In contrast with the first part of this thesis, where we have developed a new model, in this part, we use the classical model of asynchronous message-passing systems. All the basic notions (state, configuration, execution, asynchrony, ...) can be found in, e.g., [42, 63]. Another main difference with the previous part is that we have no limitations (besides being finite) on the size of the processors states.

The model represents a a system of n *asynchronous processors* in a *complete communication network*. Each communication channel between two processors is a *bidirectional asynchronous communication channel of finite capacity* C [43]. Every processor has a unique identifier and the set Π of identifiers is totally ordered. If α and β are two processor identifiers, the couple (α, β) denotes the communication channel between α and β.

A *configuration* is the vector of states of every processor and communication channel. If γ is a configuration of the system, we denote by $\gamma(\alpha)$ (resp. $\gamma(\alpha, \beta)$) the state of the processor α (resp. the communication channel (α, β)) in the configuration γ. We informally[1] define an *event* as the sending or the reception of a message at a processor or as a local transition at a processor.

Given a configuration, an event induces a transition to a new configuration. An *execution* is denoted by a sequence of configurations $(\gamma_k)_{0 \leq k < T}$, $T \in \mathbb{N} \cup \{\infty\}$ related by such transitions[2]. A *local execution* at processor λ is the sequence of states obtained as the projection of an execution on λ. If E is an execution, we denote by $E(\lambda)$ the corresponding local execution at λ.

We consider transient and crash faults only. The effect of a transient fault is to corrupt the state of some processors and/or communication channels; but it does not corrupt the memory where the program is located[3]. As usual in self-stabilization, it is assumed that all the basic services related to message transmission (in particular identifiers) are reliable. Also, we only consider the suffix of execution after the last transient fault; though crash faults may occur

[1] For a formal definition, refer to, e.g., [42, 63].
[2] For sake of simplicity, the events and the transitions are omitted.
[3] This would create Byzantine processes, and is outside of our scope.

in this suffix. This amounts to assume that the initial configuration of every execution is arbitrary.

In addition, at most f processors are prone to crash failures. We assume that at most half of the system may crash, i.e., $n \geq 2 \cdot f + 1$. A *quorum* is any set of at least $n - f$ processors. Thus, there always exists a responding majority quorum and any two quorums have a non-empty intersection.

We use the "happened-before" strict partial order introduced by Lamport [55]. In our case, we denote by $e \rightsquigarrow f$ and we say that e happens before f, or f happens after e. Note that the sentences "f happens after e" and "e does not happen before f" are not equivalent.

Finally, we fix a state-machine , and each processor has a local copy of it. A *request* corresponds to a transition of the state-machine. We assume that the machine has a predefined initial state.

8.2 The Original Paxos Algorithm

Description

In the original Paxos [56, 57], there are three roles:

- The *proposers* are responsible for receiving client requests and coordinating their execution with the other replicas.
- The *acceptors* form the memory of the system. They accept or reject the requests transmitted by the proposers. A request is ready to be executed if it is accepted by a quorum of acceptors.
- The *learners* are notified when a request is accepted by a quorum of acceptors. They then execute the request and respond to the client.

There are many ways to map these roles on the processors. For sake of clarity, we will assume that every processor can play the three roles simultaneously. Precisely, a replica always play the role of acceptor and learner. However, as we will see below, a proposer can be active or inactive, and thus, a replica can start and stop acting as a proposer.

We now describe the algorithm. Each proposer λ and each acceptor α has a ballot number t_λ and t_α respectively. These ballot numbers are simply natural numbers (unbounded). The algorithm comprises two different phases called *phase 1* and *phase 2*. We first describe the second phase. This phase, indeed, corresponds to the "normal-case operation" of Paxos. The phase 2 is triggered at λ when the proposer λ has received requests from some clients. The proposer λ then forges a sequence $proposed_\lambda$ of requests (by appending the received client requests) and broadcasts a *phase* 2 *accept* message $\langle p2a, proposed, t_\lambda \rangle$ comprising the sequence of requests and its ballot number to the acceptors.

When the acceptor α receives this message, if $t_\alpha \leq t_\lambda$, then the proposer adopts λ's ballot number, accepts the sequence of requests $proposed_\lambda$, and notifies the learners about this fact. Otherwise, $t_\alpha > t_\lambda$, the acceptor does not accept $proposed_\lambda$, and replies negatively to λ. When a learner receives the notifications for the sequence $proposed_\lambda$ from a quorum of acceptors, it executes the requests in $proposed_\lambda$ that it has not yet executed, and respond to the clients.

Hence, we see that if there is a single proposer λ, and if all the acceptors have the same ballot number as λ, then λ is able to coordinate the requests it received and make them executed by the replicas. However, since the proposer λ may crash, we need a sort of "take-over mechanism" for a new proposer to take its place. This the purpose of phase 1.

Phase 1 is triggered at μ when the processor μ begins to act as a proposer. It then creates a new value for its ballot number t_μ, and, before proceeding to phase 2, it broadcasts a *phase 1 prepare* message, $\langle p1a, t_\mu \rangle$, comprising its ballot number, to the acceptors.

When the acceptor α receives this message, if $t_\mu < t_\alpha$ (notice the strict $<$ instead of $\leq$ as in phase 2), then the acceptor α adopts the ballot number t_μ and replies positively to μ while also piggybacking the last sequence of proposals it has accepted. Otherwise, it replies negatively to μ.

If μ receives positive replies from a quorum of acceptors, then, thanks to the data they sent, μ is able to build the latest already accepted sequence of requests. It can then proceed to phase 2, and append new requests to this sequence, so that the future decisions will be coherent. If μ does not receive enough positive replies, it must create a greater value for its ballot number t_μ and re-execute the phase 1. Also, if at the end of some phase 2, μ sees that its ballot number is beaten by some acceptor's ballot number, then it also re-executes a phase 1 with a higher ballot number.

The Paxos algorithm is summed up in Alg. 10, 11 and 12. In this pseudo-code, the proposer plays the role of a distinguished learner which notifies other learners about decisions.

Algorithm 10: Paxos : Variables at processor α

```
1  (proposer)
2      client requests, queue_α : queue (read-only)
3      proposer ballot number, t^p_α : integer (init. 0)
4      proposed requests, proposed_α : requests sequence (init. empty)
5  (acceptor)
6      acceptor ballot number, t^a_α : integer (init. 0)
7      accepted requests, accepted_α = (t, seq) : t integer (init. 0), seq
       requests sequence (init. empty)
8  (learner)
9      learned requests, learned_α : requests sequence (init. empty)
10     local state, q*_α : state of the state-machine (init. initial state of the
       state-machine)
```

Remark 3. *The original formulation of Paxos [56, 57] presented the algorithm as a (partial) solution to the consensus problem. In the description above, we have presented it as a solution to the replicated state-machine, to connect more easily with the sequel. In particular, in the original formulation, there are many parallel consensus instances, the s-th of them being dedicated to decide on the s-th request to execute. We have avoided the inclusion of another counter s, and the replicas, in our presentation, decide on growing sequences of requests.*

Algorithm 11: Paxos : Prepare phase (Phase 1)

```
1  Processor λ becomes a proposer:
2      increment t_λ
3      broadcast ⟨p1a, t_λ⟩
4      collect replies R from some quorum Q
5      if all replies are positive then
6          order R according to accepted_α.t
7          proposed_λ ← accepted_α.seq the maximum in R (break ties if
           necessary)
8      else repeat phase 1
9
10 Processor α receives p1a message from λ:
11     if t_α < t_λ then adopt t_λ
12
13     reply to λ, ⟨p1b, t_α, accepted_α⟩
```

Algorithm 12: Paxos : Accept phase (Phase 2) and Decision

```
1  Once λ gets requests in queue_λ:
2      append requests to proposed_λ
3      broadcast ⟨p2a, t_λ, proposed_λ⟩
4      collect replies R from some quorum Q
5      if all replies are positive then
6          broadcast ⟨dec, t_λ, proposed_λ⟩
7      else proceed to phase 1
8
9  Processor α receives p2a or dec message from λ:
10     if t_α ≤ t_λ then
11         accept (t_λ, proposed_λ)
12         if it is a dec message then
13             learn proposed_λ
14             update q*_α by executing the new requests
15     if it is a p2a message then
16         reply to λ, ⟨p2b, t_α, accepted_α⟩
```

Paxos is a Partial Solution

Lamport has shown [57] that Paxos always guarantee linearizability. In our case, it means that if two sequences of requests are decided on by some replicas, then one of them is the prefix of the other. Indeed, if there is a unique proposer, we have seen there are no conflicts: the requests are ordered by the proposer and the replicas decide on the requests built by the proposer. The difficulties occur when the proposer λ awakes and execute phase 1. In this phase, λ retrieves information about the past decisions from a quorum of acceptors. If it were to choose a wrong sequence of requests to start with, then it could compromise future decisions. However, this does not happen because, roughly speaking, a sequence of requests may be decided on, only when a quorum of acceptors have accepted it. The fact that two quorums have a non-empty intersection implies that, at the end of phase 1, the proposer λ actually gets the correct information about the past decisions.

On the other hand, Paxos is a partial solution in the sense that the liveness property is not guaranteed. This is not a surprise since state-machine replication is closely related to the consensus problem, and [47] has shown that consensus is impossible in this environment. In Paxos, liveness is not achieved because many proposers may be active at the same time. This can happen because, the take-over mechanism implemented in phase 1, is usually triggered when a replica detects the crash of the proposer. But detecting crashes is unreliable. Two active proposers may compete in having the greatest ballot number to be able to coordinate the requests. Thus, they never execute phase 2, the system is stalled. Such a scenario, though, is unlikely in practice.

8.3 How to Make Paxos Self-Stabilizing ?

In this section, we informally present how to make Paxos (practically) self-stabilizing. We first need to examine what would be the effects of a transient fault on the original Paxos algorithm. Obviously, it can corrupt the local copies of the state-machine (variables q^*_α, $learned_\alpha$ or $proposed_\alpha$). However, as stated in Chap. 7, this is not the worse issue. If the replicas can still agree on sequences of requests, they can reset their local state-machines and re-execute the common sequence of requests to be up-to-date.

The core of the Paxos algorithm relies on a clever management of the ballot numbers. In any concrete implementation of Paxos, these ballot numbers are integers bounded by a large (but finite) constant $2^\mathfrak{b}$. A corruption that sets some of these ballot numbers to the maximum value will permanently hinders the system. For instance, the proposers will not be able to get a higher ballot number, which prevents them to succeed in any phase 1 or phase 2.

Our approach consists in resetting the ballot numbers only if necessary. If we call epoch the segment of execution between two such resets, the goal is to force the existence of a practically infinite epoch (i.e. an epoch containing a causal event chain of length greater than $2^\mathfrak{b}$) during which the ballot numbers start with low values. During such an epoch, everything looks like an initialized Paxos execution.

We now explain how to implement such resets. The crucial property of the ballot numbers are the fact that, when the proposer λ sees a collection of ballot numbers from a quorum of acceptors, λ is able to create a ballot

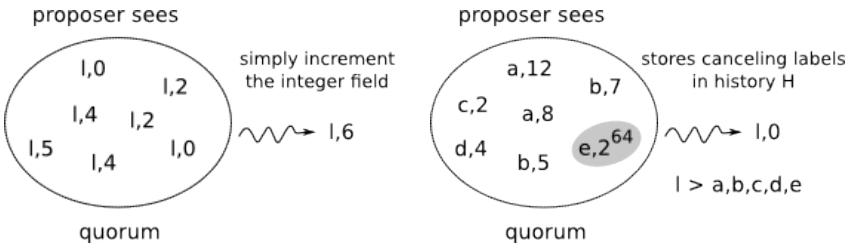


Figure 8.1: Incrementing ballot numbers – (left) the acceptors and the proposer have the same label, (right) they have different labels.

number greater than all these ballot numbers. Obviously, unbounded natural numbers are the most intuitive structure which offers this possibility. Note that the whole problem relies on the fact that bounded natural numbers are not suitable for such a task, since it is impossible to create a value greater than the maximum value.

Anyway, the structure of natural numbers is not necessary. Indeed, a *bounded labeling scheme*, as in [4], offers this possibility too. Informally, it consists in a finite set $\mathcal{L}$ of labels, along with a comparison operator $\prec$ and an increment function ν. The increment function maps any (bounded) set H of labels to a label $\nu(H)$ that is greater than all the labels in H, according to the relation $\prec$. See Chap. 9, Sec. 9.3 for details on how to implement a bounded labeling scheme.

To see how we can use such a a scheme to implement clean resets, let's focus on a toy example where a unique replica plays the role of the proposer. We then redefine a ballot number as a couple (l, t) where l is a label, and t is a bounded integer. These ballot numbers are compared lexicographically. If the proposer and the acceptors all use the same label l, as depicted in Fig. 8.1, then, to produce a greater ballot number, the proposer simply increments the integer field. This corresponds exactly to the use of ballot numbers in Paxos. But, since the initial configuration is arbitrary, the replicas may use different labels, or have an integer value close to the maximum $2^{\mathfrak{b}}$, as in Fig. 8.1. In that case, the proposer stores in a history H^{cl} every *canceling label*, i.e., the labels that are not beaten by the proposer's label, use the increment function to produce a greater label $\nu(H^{cl})$, and reset the integer field to zero. Doing so, in the sequel, the acceptors will adopt this new label and start with low integer values: the system will then behave as an initialized Paxos instance. Therefore, if the proposer collects enough information in its history H^{cl} about the labels present in the system, then the proposer can produce a greatest label, and from there on, the system will behave as in the original Paxos. Note that the proposer does not actually need to collect all the initially hidden labels in the system. If a canceling label remains hidden for a practically infinite period of time, then it does no harm to the system. If this label shows up, the proposer notices it, and produces a greater label.

Unfortunately, one cannot require that a unique replica plays the role of a proposer because of crash failures. If we use the previous technique in case of many proposers, then the proposers will compete in trying to get the greatest label. This prevents the system to reach a practically infinite period during

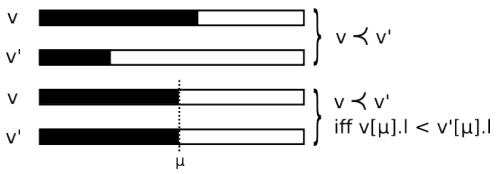


Figure 8.2: Comparison of tags - Invalid entries are darkened.

which the label is stable, and the integer fields behave as the ballot numbers in the original Paxos.

To avoid the interferences between the proposers, we introduce slightly more intricate data structure called *tag*. Roughly speaking, a tag v is a vector indexed by the proposers identifiers. The entry μ of the tag v contains a label field, and a "canceling field" used to notifies about possible canceling labels or overflows of integer counters. The basic idea is that the proposer whose identifier is μ is the *unique* proposer producing new labels in the entry μ of the tags. Each proposer μ then tries, as in the single proposer case, to get a greatest label for the entry μ only.

Similarly to the single proposer case, we will define a ballot number as a couple (v, t) where v is a tag, and t a bounded integer. To compare such ballot numbers, it remains to define how to compare tags. To do so, we assume that the identifiers are totally ordered. The basic idea is that if the two proposers $\lambda < \mu$ manage to find greatest labels in their respective entries, then they should both use the entry $\lambda = \min(\lambda, \mu)$ as their "active entry". Put another way, if the active entry of a tag refers to the first non-canceled entry, then we compare tags as in Fig. 8.2. More precisely, if the active entry of the tag v is located after the active entry of the tag v', then $v \prec v'$. If the active entries refer to the same place, then we simply compare the corresponding labels.

Thanks to this comparison relation, a proposer will seek to adopt tags which have the "leftmost" active entry. Fig. 8.3 illustrates how the tags propagate among many proposers. Time goes downwards and the arrows represent exchanges of messages. The processors have, as ballot numbers, couples (v, t) where v is a tag, and t a bounded integer. A first proposer uses a tag v and manages to impose v to the acceptors. Meanwhile, a second proposer awakes with a greater tag v', and the acceptors adopt it. Then the former proposer also adopts the tag v'.

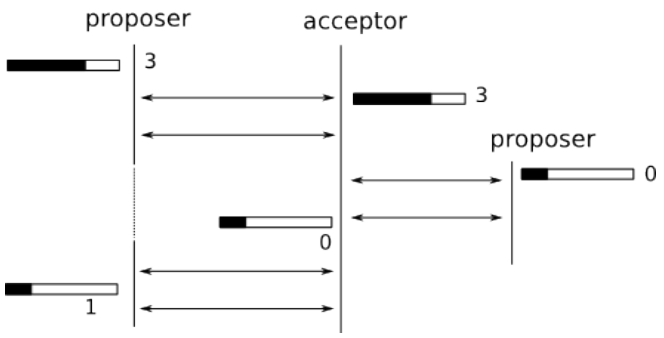


Figure 8.3: Paxos message flow – ballot number = (tag, integer)

Chapter 9

Practically Self-Stabilizing Replicated State-Machine

In this chapter, we give a more detailed description of our algorithm.

9.1 Data structures

Given a positive integer $\mathfrak{b}$, a $\mathfrak{b}$-bounded integer, or simply a bounded integer, is any non-negative integer less than or equal to $2^{\mathfrak{b}}$. A *finite labeling scheme* is a 4-tuple $\overline{\mathcal{L}} = (\mathcal{L}, \prec, d, \nu)$ where $\mathcal{L}$ is a finite set whose elements are called *labels*, $\prec$ is a partial relation on $\mathcal{L}$ that is irreflexive ($l \not\prec l$) and antisymmetric ($\nexists (l, l')\ l \prec l' \wedge l' \prec l$), d is an integer, namely the *dimension* of the labeling scheme, and ν is the *label increment function*, i.e., a function that maps any finite set A of at most d labels to a label $\nu(A)$ such that for every label l in A, we have $l \prec \nu(A)$. We denote the reflexive closure of $\prec$ by $\preccurlyeq$. The definition of a finite labeling scheme imposes that the relation $\prec$ is not transitive. Hence, it is not a preorder relation. Given a label l, a *canceling label* for l is a label cl such that $cl \not\preccurlyeq l$. See Sec. 9.3 for a concrete construction of a finite labeling scheme of any dimension.

A *tag* is a vector $v[\mu] = (l\ cl)$ where $\mu \in \Pi$ is a processor identifier, l is a label, cl is either the null symbol $\perp$, the overflow symbol ∞ or a canceling label for l. The entry μ in v is said to be *valid* when the corresponding canceling field is null, $v[\mu].cl = \perp$. If v has at least one valid entry, we denote by $\chi(v)$ the *first valid entry of* v, i.e., the smallest identifier μ such that $v[\mu]$ is valid. If v has no valid entry, we set $\chi(v) = \omega$ where ω is a special symbol (not in Π). Given two tags v and v', $v \prec v'$ if either $\chi(v) > \chi(v')$ or $\chi(v) = \chi(v') = \mu \neq \omega$ and $v[\mu].l < v'[\mu].l$ (see Fig. 8.2 in Chap. 8, Sec. 8.3). We write $v \simeq v'$ when $\chi(v) = \chi(v') = \mu$ and $v[\mu] = v'[\mu]$. We write $v \preccurlyeq v'$ when either $v \prec v'$ or $v \simeq v'$.

A *fifo label history* H of size d, is a vector of size d of labels along with an operator $+$ defined as follows. Let $H = (l_1, \ldots, l_d)$ and l be a label. If l does not appear in H, then $H + l = (l, l_1, \ldots, l_{d-1})$, otherwise $H + l = H$. We define the *tag storage limit* K and the *canceling label storage limit* K^{cl} by $K = n + C\frac{n(n-1)}{2}$ and $K^{cl} = (n+1)K$.

9.2 The Algorithm

In this section, we describe the practically self-stabilizing Paxos algorithm. In its essence, our algorithm is enhances the Paxos scheme with the tag system in order to cope with overflows.

The variables are presented in Alg. 13. The clients are not modeled here; we simply assume that each active proposer α can query a stream $queue_\alpha$ to get a client request to propose. The variables are divided in three sections corresponding to the different Paxos roles: proposer, acceptor, learner. In each section, some variables are marked as Paxos variables while the others are related to the tag system.

The message flow is similar to Paxos. When a proposer λ becomes active, it executes a prepare phase (phase 1), trying to recruit a majority of acceptors. An acceptor α is recruited if the proposer ballot number is (strictly) greater than its own ballot number. In this case, it adopts the ballot number. It also replies (positively or negatively) to the leader with its latest accepted sequence of requests $accepted_\alpha$ along with the corresponding (integer) ballot number. After recruiting a quorum of acceptors, the proposer λ records the latest sequence (w.r.t. the associated integer ballot numbers) of requests accepted by them in its variable proposed $proposed_\lambda$. If this phase 1 is successful, the proposer λ can execute accept phases (phase 2) for each request received in $queue_\lambda$. For each such request r, the proposer λ appends r to its variable $proposed_\lambda$, and tell the acceptors to accept $proposed_\lambda$. An acceptor accepts the proposal $proposed_\lambda$ when the two following conditions are satisfied: (1) the proposer's ballot number is greater than or equal to its own ballot number, and (2) if the ballot integer associated with the lastly accepted proposal is equal to the proposer's ballot integer, then $proposed_\lambda$ is an extension of the lastly accepted proposal. Roughly speaking, this last condition avoids the acceptor to accept an older (hence shorter) sequence of request. In any case, the acceptor replies (positively or negatively) to the proposer. The proposer λ plays the role of a special learner in the sense that it waits for positive replies from a quorum of acceptors, and, sends the corresponding decision message. The decision procedure when receiving a decision message is similar to the acceptation procedure (reception of a $p2a$ message), except that if the acceptor accepts the proposal, then it also learns (decides on) this proposal and execute the corresponding new requests.

We now describe the treatment of the variables related to the tag system. Anytime a processor α (as an acceptor, learner or proposer) with tag v_α receives a message with a tag v', it updates the canceling label fields before comparing them, i.e., for any μ, if $v_\alpha[\mu].l$ (or $v_\alpha[\mu].cl$) is a label that cancels $v'[\mu].l$, or $v_\alpha[\mu].cl = \infty$ is the overflow symbol, then the field $v'[\mu].cl$ is updated accordingly[1], and vice versa. Also, if the processor α notices an overflow in its own variables (e.g. its ballot integer, or one of the request sequence variables, has reached the upper bound), it sets the overflow symbol ∞ in the canceling field of the first valid entry of the tag. If after such an update, the label $v_\alpha[\alpha].l$ is canceled, then the corresponding canceling label is added to H_α^{cl} as well as the label $v_\alpha[\alpha].l$, and $v_\alpha[\alpha].l$ is set to the new label $\nu(H_\alpha^{cl})$ created from

[1] i.e., the field $v'[\mu].cl$ is set to $v_\alpha[\mu].(l \text{ or } cl)$. In case, there is a canceling label and the overflow symbol, the canceling label is preferred.

the labels in H_α^{cl} with the label increment function. The purpose of H_α^{cl} is to record enough canceling labels for the proposer to produce a greatest label. In addition, if, after the update, it appears that $v_\alpha \preccurlyeq v'$, then α adopts the tag v', i.e., it copies the content of the first valid entry $\mu = \chi(v')$ of v' to the same entry in v_α (assuming $\mu < \alpha$). Doing so, it also records the previous label in v_α in the label history $H_\alpha[\mu]$. If there is a label in $H_\alpha[\mu]$ that cancels $v_\alpha[\mu].l$, then the corresponding field is updated accordingly. The purpose of $H_\alpha[\mu]$ is to avoid cycle of labels in the entry μ of the tag. Recall that the comparison between labels is not a preorder. In case $\mu = \alpha$, then α uses the label increment function on H_α^{cl} to produce a greater label as above.

We say that there is an *epoch change* in the tag v_λ if either the first valid entry $\chi(v_\lambda)$ has changed, or the first valid entry has not changed but the corresponding label has changed. Whenever there is an epoch change in the tag v_λ the processor cleans the Paxos related variables. For a proposer λ, this means that the proposer ballot integer t_λ^p is reset to zero, the proposed requests $proposed_\lambda$ to the empty sequence; in addition, the proposer proceeds to a new prepare phase. For an acceptor (and learner) α, this means that the acceptor ballot integer is reset to zero, the sequences $accepted_\alpha$ and $learned_\alpha$ are reset to the empty sequence, and the local state q_α^* is reset to the predefined initial state of the state-machine.

The pseudo-code in Algorithms 14 and 15 sums up the previous description. Note that, the predicate $(v_\alpha, t_\alpha) < (v_\lambda, t_\lambda)$ (resp. $(v_\alpha, t_\alpha) \leq (v_\lambda, t_\lambda)$) means that either $v_\alpha \prec v_\lambda$, or $v_\alpha \simeq v_\lambda$ and $t_\alpha < t_\lambda$ (resp. $v_\alpha \prec v_\lambda$, or $v_\alpha \simeq v_\lambda$ and $t_\alpha \leq t_\lambda$).

Remark 4. *Note that, in our algorithm, the replicas agree on growing sequences of requests, of length at most $2^{\mathfrak{b}}$. We do not focus on optimizations for the sake of simplicity. Yet, a means to control the length of the sequences would be to replace a prefix of request sequence by the state reached from the initial state when applying the prefix. Then the replicas can agree on (possibly conflicting) states by the latest found in a quorum.*

9.3 Bounded Labeling Scheme

In this section, we give a concrete implementation of the bounded labeling scheme used in our algorithm. This construction comes from [4].

First, consider the set of integers $X = \{1, 2, ..., d^2 + 1\}$. We define the set $\mathcal{L}$ to be the set of every tuple (z, A) where $z \in X$ is the *sting*, and $A \subset X$ with $|A| \leq d$ is called the *antistings*. The relation $\prec$ is defined as follows

$$l = (z, A) \prec l' = (z', A') \Leftrightarrow (z \in A') \wedge (z' \notin A) \tag{9.1}$$

The function ν is defined as follows. Given r labels (s_1, A_1), ..., (s_r, A_r) with $r \leq d$, the label $\nu(l_1, \ldots, l_r) = (s, A)$ is given by

$$s = \text{ any element in } X - (A_1 \cup \cdots \cup A_r) \tag{9.2}$$

$$A = \{s_1, \ldots, s_r\} \tag{9.3}$$

The function is well-defined since $r \leq d$ and $|A_1 \cup \cdots \cup A_r| \leq d^2 < |X|$. In addition, for every i, we have $s \notin A_i$ and $s_i \in A$, thus $(s_i, A_i) \prec (s, A)$.

Algorithm 13: Variables at processor α

(tag system)
    v_α : tag
    canceling label history, H_α^{cl} : fifo history of size $(K+1)K^{cl}$
    for each $\mu \in \Pi$, label history, $H[\mu]$: fifo history of size K
(proposer)
    client requests, $queue_\alpha$: queue (read-only)
    [Paxos] proposer ballot integer, t_α^p : bounded integer
    [Paxos] proposed requests, $proposed_\alpha$: requests sequence of size $\leq 2^{\mathfrak{b}}$
(acceptor)
    [Paxos] acceptor ballot integer, t_α^a : bounded integer
    [Paxos] accepted requests, $accepted_\alpha = (t, seq)$: t bounded integer, seq requests sequence of size $\leq 2^{\mathfrak{b}}$
(learner)
    [Paxos] learned requests, $learned_\alpha$: requests sequence of size $\leq 2^{\mathfrak{b}}$
    [Paxos] local state, q_α^* : state of the state-machine

Algorithm 14: Prepare phase (Phase 1)

```
1  Processor λ becomes a proposer:
2      increment t_λ
3      if t_λ reaches 2^𝔟 then
4          set v_λ[χ(v_λ)].cl to ∞
5          update the entry v_λ[λ] with H^cl if it is invalid
6          clean the proposer Paxos variables
7      broadcast ⟨p1a, v_λ, t_λ, λ⟩
8      collect replies R from some quorum Q
9      update (if necessary) the tag v_λ and the label histories
10     if no epoch change in v_λ and all replies are positive then
11         order R with lexicographical order (accepted_α.t, |accepted_α.seq|)
12         proposed_λ ← accepted_α.seq the maximum in R (break ties if
           necessary)
13         if proposed_λ has reached max length then
14             set v_λ[χ(v_λ)].cl to ∞
15             update the entry v_λ[λ] with H^cl if it is invalid
16             clean the Paxos variables
17             repeat phase 1
18     else
19         if epoch change in v_λ then
20             clean the Paxos variables
21         repeat phase 1
22 Processor α receives p1a message from λ:
23     update canceling fields in (v_α, v_λ)
24     if (v_α, t_α) < (v_λ, t_λ) then
25         adopt v_λ, t_λ
26     if epoch change in v_α then
27         clean Paxos variables
28     reply to λ, ⟨p1b, v_α, t_α, accepted_α, α⟩
```

Algorithm 15: Accept phase (Phase 2) and Decision

```
1  Once λ gets requests in queue_λ:
2      append requests to proposed_λ
3      broadcast ⟨p2a, v_λ, t_λ, proposed_λ⟩
4      collect replies R from some quorum Q
5      update (if necessary) the tag v_λ and the label histories
6      if no epoch change in v_λ and all replies are positive then
7          broadcast ⟨dec, v_λ, t_λ, proposed_λ⟩
8      else
9          if epoch change in v_λ then clean the Paxos variables
10
11         proceed to phase 1
12 Processor α receives p2a or dec message from λ:
13     update canceling fields in (v_α, v_λ)
14     if (v_α, t_α) ≤ (v_λ, t_λ) then
15         adopt v_λ, t_λ
16         if epoch change in v_α then clean the Paxos variables
17
18         if accepted_α.t < t_λ or accepted_α.seq is a prefix of proposed_λ
           then
19             accept (t_λ, proposed_λ)
20             if it is a dec message then
21                 learn proposed_λ
22                 update q*_α by executing the new requests
23     if it is a p2a message then
24         reply to λ, ⟨p2b, v_α, t_α, accepted_α, α⟩
```

Chapter 10

Analysis

In this chapter, we prove the main properties of our algorithm. We first present some basic and useful lemmas in Sec. 10.1. In Sec. 10.2, we prove that there exists a practically infinite epoch at at least one proposer (Prop. 25). This epoch is safe in a specific sense (Def. 10). In Sec. 10.3, we use the previous result to exhibit a globally defined segment of execution (Def. 20) and prove that, within this segment of execution, the safety property is ensured (Prop. 27), in the sense that, if two sequences of requests are decided on within this segment, then one of them is the prefix of the other. Finally, in Sec. 10.4, we exhibit a simple, but non-trivial, self-stabilizing implementation of a failure detector that works under a partial synchrony assumption.

10.1 Basics

The pigeon-hole principle is a well-known combinatorial argument used to prove the existence of an object.

Lemma 20 (Pigeon-hole Principle)**.** *Consider a sequence $u = (u^i)_{1 \leq i \leq N}$ such that $\forall 1 \leq i \leq N, u^i \in \{0,1\}$, and $N = (n+1)m$ for some $n, m \in \mathbb{N} - \{0\}$. Assume that the cardinal of $\{i \mid u^i = 1\}$ is less than or equal to n. Then there exists $1 \leq i_0 \leq N$ such that for every $i_0 \leq i \leq i_0 + m - 1$, $u^i = 0$.*

Proof. Divide the sequence u in successive subsequences σ^j, $1 \leq j \leq n+1$ such that each σ^j length is m. If for every $1 \leq j \leq n+1$, the sequence σ^j contains at least one 1, then the number of 1 appearing in u is at least $n+1$, which leads to a contradiction. Hence, there is some j_0 such that the sequence σ^j only contains 0. □

Since the initial configuration of an execution is arbitrary, we do not know the initial values of the states, the messages, and, in particular, the label values they contain. The following lemma gives a bound on the maximum number of different label values present in a configuration. Given any configuration γ of the system and any processor identifier μ, let $S(\gamma)$ and $S^{cl}(\mu, \gamma)$ be two sets as follows. The set $S(\gamma)$ is the set of every tag present either in a processor memory or in some message in a communication channel, in the configuration γ. The set $S^{cl}(\mu, \gamma)$ denotes the collection of labels l such that either l is the

value of the label field $x[\mu].l$ for some tag x in $S(\gamma)$, or l appears in the label history $H_\alpha[\mu]$ of some processor α, in the configuration γ.

Lemma 21 (Storage Limits)**.** *For every configuration γ and every identifier μ, we have $|S(\gamma)| \leq K$ and $|S^{cl}(\mu, \gamma)| \leq K^{cl}$. In particular, the number of label values $x[\mu].l$ with x in $S(\gamma)$ is less than or equal to K.*

Proof. Consider a configuration γ. For each processor α, there is one tag value in the processor state $\gamma(\alpha)$ of α. For each communication channel (α, β), there are at most C different messages in the channel state $\gamma(\alpha, \beta)$; each of them have one tag. Hence, the maximum number of tags present in the configuration γ is n plus C times the number of communication channels. The network being complete, the number of communication channels is $C\frac{n(n-1)}{2}$, thus we have $K \geq |S(\gamma)|$. For every α, the maximum size of the history $H_\alpha[\mu]$ is K. Hence, the size of $S^{cl}(\mu, \gamma)$ is bounded above by K (labels $x[\mu].l$ for x in $S(\gamma)$) plus K times the number of processors (labels from $H_\alpha[\mu]$ for every processor α), i.e., $(n+1) \cdot K = K^{cl}$. □

10.2 Tag Stabilization

Definitions

As state in the introduction, our approach relies on a clean reset mechanism of the Paxos related variables. These resets occur when the corresponding tag undergoes a change of label or a change of active entry. We refer to such events as *interrupts*. The following definition classify the possible kinds of interrupts.

Definition 8 (Interrupt)**.** *Let λ be any processor (as a proposer, or an acceptor) and consider a local execution segment $\sigma = (\gamma_k(\lambda))_{k_0 \leq k \leq k_1}$ at λ. We denote by v_λ^k the λ's tag in $\gamma_k(\lambda)$. We say that an interrupt has occurred at position k in the local subexecution σ when one of the following happens*

- *$\mu < \lambda$, type $[\mu, \leftarrow]$: the first valid entry moves to μ such that $\mu = \chi(v_\lambda^{k+1}) < \chi(v_\lambda^k)$, or the first valid entry does not change but the label does, i.e., $\mu = \chi(v_\lambda^{k+1}) = \chi(v_\lambda^k)$ and $v_\lambda^k[\mu].l \neq v_\lambda^{k+1}[\mu].l$.*
- *$\mu < \lambda$, type $[\mu, \rightarrow]$: the first valid entry moves to μ such that $\mu = \chi(v_\lambda^{k+1}) > \chi(v_\lambda^k)$.*
- *type $[\lambda, \infty]$: the first valid entry is the same but there is a change of label in the entry λ due to an overflow of one of the Paxos variables; we then have $\chi(v_\lambda^{k+1}) = \chi(v_\lambda^k) = \lambda$ and $v_\lambda^k[\lambda].l \neq v_\lambda^{k+1}[\lambda].l$.*
- *$[\lambda, cl]$: the first valid entry is the same but there is a change of label in the entry λ due to the canceling of the corresponding label; we then have $\chi(v_\lambda^{k+1}) = \chi(v_\lambda^k) = \lambda$ and $v_\lambda^k[\lambda].l \neq v_\lambda^{k+1}[\lambda].l$.*

For each type $[\mu, *]$ $(\mu \leq \lambda)$ of interrupt, we denote by $|[\mu, *]|$ the total number (possibly infinite) of interrupts of type $[\mu, *]$ that occur during the local execution segment σ.

If there is an interrupt like $[\mu, \leftarrow]$, $\mu < \lambda$, occurs at position k, then necessarily there is a change of label in the field $v_\lambda[\mu].l$ (due to the adoption of received tag). In addition, the new label l' is greater than the previous label l, i.e., $l \prec l'$. Also note that, if $\chi(v_\lambda^k) = \lambda$, the proposer λ never copies the content of the entry λ of a received tag, say v', to the entry λ of its tag, even if $v_\lambda^k[\lambda].l \prec v'[\lambda].l$. New labels in the entry λ are only produced with the label increment function applied to the union of the current label and the canceling label history H_λ^{cl}.

It is now possible to formally define an epoch as a local execution at some processor between two interrupts.

Definition 9 (Epoch)**.** *Let λ be a processor. An epoch σ at λ is a maximal*[1] *local execution segment at λ such that no interrupts occur at any position in σ except for the last position. By the definition of an interrupt, all the tag's values within a given epoch σ at λ have the same first valid entry, say μ, and the same corresponding label, i.e., for any two processor states that appear in σ, the corresponding tag values v and v' satisfies $\chi(v) = \chi(v') = \mu$ and $v[\mu].l = v'[\mu].l$. We denote by μ_σ and l_σ the first valid entry and the corresponding label common to all the tag values in σ.*

If there is an epoch σ at processor λ such that $\mu_\sigma = \lambda$ and λ has produced the label l_σ, then necessarily, at the beginning of σ, the Paxos variables have been reset. However, other processors may already be using the label l_σ with, for example, arbitrary ballot integer value. Such an arbitrary value may be the cause of the overflow interrupt at the end of σ. The definition of a h-safe epoch ensures that the epoch is truly as long as counting from h to $2^{\mathfrak{b}}$.

Definition 10 (h-Safe Epoch)**.** *Consider an execution E and a processor λ. Let Σ be an execution segment in E such that the local execution segment $\sigma = \Sigma(\lambda)$ is an epoch at λ. Let γ^* be the configuration of the system right before Σ, and h be a bounded integer. The epoch σ is said to be* h-safe *when the interrupt at the end of σ is due to an overflow of one of the Paxos variables. In addition, for every processor α (resp. communication channel (α, β)), for every tag x in $\gamma^*(\alpha)$ (resp. $\gamma^*(\alpha, \beta)$), if $x[\mu_\sigma].l = l_\sigma$ then any corresponding integer variables (ballot integers, or lengths of request sequences) have values less than or equal to h.*

Results

Since each processor λ keeps looking for a greatest label in the entry λ of its tag, the first valid entry of its tag is eventually always located before the entry λ.

Lemma 22. *Let λ be any processor. Then the first valid entry of its tag is eventually always located at the left of the entry indexed by λ, i.e., $\chi(v_\lambda) \leq \lambda$.*

Proof. This comes from the fact that whenever the entry $v_\lambda[\lambda]$ is invalid, the processor λ produces a new label in $v_\lambda[\lambda]$. Once $\chi(v_\lambda) \leq \lambda$, every subsequent tag values is obtained as above, or by copying the content of a valid entry $\mu < \lambda$ of some tag to the entry $v_\alpha[\mu]$. Hence the first valid entry remains located before the entry λ. □

[1] For the inclusion of local execution segments.

Thanks to this lemma, for every processor λ, it is now assumed, unless stated explicitly, that the entry $\chi(v_\lambda)$ is always located before the entry λ, i.e., $\chi(v_\lambda) \leq \lambda$.

The processor λ is not responsible for the creation of labels in any entry $\mu < \lambda$. Yet, since the label comparison operator is not transitive, it is possible for the label field in the entry μ to follow a cycle of labels. Lem. 23 gives information about the length of such cycles. Indeed, since the label history $H_\lambda[\mu]$ records the latest values that were present in the label field of the entry μ, the cycle length must be greater than the history size. The history size is chosen so that the proposer μ must have produced a label meanwhile.

Lemma 23 (Cycle of Labels)**.** *Consider an execution segment E, a processor λ and an entry $\mu < \lambda$ in the tag variable v_λ. The label value in $v_\lambda[\mu].l$ can change during E and we denote by $(l^i)_{1 \leq i \leq T+1}$ for the sequence of successive distinct label values that are taken by the label $v_\lambda[\mu].l$ in the entry μ during E. We assume that the first T labels $l^1, \ldots, l^T$ are different from each other, i.e., for every $1 \leq i < j \leq T$, $l^i \neq l^j$. If $T > K$, then at least one of the label l^i has been produced[2] by the processor μ during E. If $T \leq K$ and $l^{T+1} = l^1$, then when the processor λ adopts the label l^{T+1} in the entry μ of its tag v_λ, the entry μ becomes invalid.*

Proof. First note that a processor adopts a new label in the entry μ of one of its tag, only when the old label is less than the new label. Hence, we have for every $1 \leq i \leq T$, $l^i \prec l^{i+1}$ and, in particular, if $l^1 = l^{T+1}$, $l^2 \not\preccurlyeq l^{T+1}$. Assume $T > K$. Since in every configuration there is at most K tags in the system, since μ is the only source of labels in the entry μ, and since λ records the last K label values in the history $H_\lambda[\mu]$, the fact that λ has seen more than K different label values in the entry μ is possible only if μ has produced at least one label during E. If $T \leq K$ and $l^1 = l^{T+1}$, i.e., there is a cycle of length T, then when λ adopts the label $l^{T+1} = l^1$, the label history $H_\lambda[\mu]$ contains the whole sequence $l^1, \ldots, l^T$ since its size is K. Hence, λ sees the label l^2 that cancels the label l^{T+1}, and the entry μ becomes invalid. □

Thanks to this control on the length of the cycles, we can compute a bound on the number of interrupts which induce a label change in the entry μ.

Lemma 24 (Counting the Interrupts)**.** *Consider an infinite execution E_∞ and let λ be a processor identifier such that every processor $\mu < \lambda$ produces labels finitely many times. Consider an identifier $\mu < \lambda$ and any processor $\rho \geq \lambda$. Then, the local execution $E_\infty(\rho)$ at ρ induces a sequence of interrupts such that $|[\mu, \leftarrow]| \leq R_\mu = (J_\mu + 1) \cdot (K + 1) - 1$ where J_μ is the number of times the processor μ has produced a label since the beginning of the execution.*

Proof. We denote by $(v_\rho^k)_{k \in \mathbb{N}}$ the sequence of ρ's tag (v_ρ) values appearing in the local execution $E_\infty(\rho)$. Assume on the contrary that $|[\mu, \leftarrow]|$ is greater than R_μ. Note that after an interrupt like $[\mu, \leftarrow]$, the first valid entry $\chi(v_\rho)$ is equal to μ. In particular, the entry μ is valid after such interrupts. Also, the label value in the entry $v_\lambda[\mu].l$ does not change after an interrupt like $[\mu, \rightarrow]$.

[2] Precisely, it has invoked the label increment function to update the entry μ of its tag v_μ.

We define an increasing sequence of integers $(f(i))_{1\leq i\leq R_\mu+1}$ such that the i-th interrupt like $[\mu,\leftarrow]$ occurs at $f(i)$ in the sequence $(v_\rho^k)_{k\in\mathbb{N}}$. The sequence $l^i = v_\rho^{f(i)+1}[\mu].l$ is the sequence of distinct labels successively taken by $v_\rho[\mu].l$. We have $l^i \prec l^{i+1}$ for every $1 \leq i \leq R_\mu$.

Divide the sequence $(l^i)_{1\leq i\leq R_\mu+1}$ in successive segments u^j, $1 \leq j \leq J_\mu+1$, of size $K+1$ each. For any j, if all the $K+1$ labels in u^j are different, then, by Lem. 23, the processor μ has produced at least one label. Since the processor μ produces labels at most J_μ many times, there is some sequence u^{j_0} within which some label appears twice. In other words, in u^{j_0} there is a cycle of length less than or equal to K. By Lem. 23, this implies that the entry μ becomes invalid after an interrupt like $[\mu,\leftarrow]$; this is a contradiction. □

We are now able to prove the main proposition of this section, i.e., the existence of a 0-safe epoch at a processor.

Proposition 25 (Existence of a 0-Safe Epoch)**.** *Consider an infinite execution E_∞ and let λ be a processor such that every processor $\mu < \lambda$ produces labels finitely many times. We denote by $|\lambda|$ the number of identifiers $\mu \leq \lambda$, J_μ for the number of times a proposer $\mu < \lambda$ produces a label and we define*

$$T_\lambda = (\sum_{\mu<\lambda} R_\mu + 1) \cdot (|\lambda| + 1) \cdot (K^{cl} + 1) \cdot (K + 1) \tag{10.1}$$

where $R_\mu = (J_\mu+1)\cdot(K+1)-1$. Assume that there are more than T_λ interrupts at processor λ during E_∞ and consider the concatenation $E_c(\lambda)$ of the first T_λ epochs, $E_c(\lambda) = \sigma^1 \dots \sigma^{T_\lambda}$. Then $E_c(\lambda)$ contains a 0-safe epoch.

Proof. By Lem. 24, we have $\sum_{\mu<\lambda} |[\mu,\leftarrow]| \leq \sum_{\mu<\lambda} R_\mu$ in the local execution $E_\infty(\lambda)$, a fortiori in the execution $E_c(\lambda)$. By the pigeon-hole principle, there must be a local execution segment $E_1(\lambda) = \sigma^i \dots \sigma^{i+X-1}$ in $E_c(\lambda)$, where $X = (|\lambda| + 1) \cdot (K^{cl} + 1) \cdot (K + 1)$, that contains only interrupts like $[\mu,\rightarrow]$, $[\lambda,\infty]$ or $[\lambda, cl]$. Naturally, the number of interrupts like $[\mu,\rightarrow]$ in $E_1(\lambda)$ is less than or equal to $|\lambda|$. Hence, another application of the pigeon-hole principle gives a local execution segment $E_2(\lambda) = \sigma^j \dots \sigma^{j+Y-1}$ in $E_1(\lambda)$ where $Y = (K^{cl} + 1) \cdot (K + 1)$ that contains only interrupts like $[\lambda,\infty]$ or $[\lambda, cl]$.

Assume first that within $E_2(\lambda)$, there is a execution segment $E_3(\lambda) = \sigma^k \dots \sigma^{k+Z-1}$ where $Z = K + 1$ in which there are only interrupts like $[\lambda,\infty]$. Since $K + 1$ is less than or equal to the size of the canceling label history[3], we have $l_{\sigma^k}, \dots, l_{\sigma^{h-1}} \prec l_{\sigma^h}$, for every $k < h < k + Z$. In particular, all the labels $l_{\sigma^k}, \dots, l_{\sigma^{k+z-1}}$ are different. Since $Z = K + 1$ and since there is at most K tags in a given configuration, there is necessarily some $k \leq h < k + Z$ such that the label l_{σ^h} does not appear[4] in the configuration γ^* that corresponds to the last position in σ^{h-1}. Also, by construction, we have $\mu_{\sigma^h} = \lambda$ and σ^h ends with an interrupt like $[\lambda,\infty]$. Hence, σ^h is 0-safe.

Now, assume that there is no execution segment E_3 in E_2 as in the previous paragraph. This means that if we look at the successive interrupts that occur during $E_2(\lambda)$, between any two successive interrupts like $[\lambda, cl]$, there is at most

[3]Recall that the canceling label history also records the label produced in the entry λ.

[4]Note that λ is the only processor to produce labels in entry λ, so during the execution segment that corresponds to an epoch σ^h at λ, the set of labels in the entry λ of every tag in the system is non-increasing.

K interrupts like $[\lambda, \infty]$. Since the length of $E_2(\lambda)$ is $(K^{cl}+1)\cdot(K+1)$, there must be at least $K^{cl}+1$ interrupts like $[\lambda, cl]$. Let $E_4(\lambda)$ be the local execution segment that starts with the epoch associated with the first interrupt like $[\lambda, cl]$ and ends with the epoch associated with the interrupt $[\lambda, cl]$ numbered K^{cl}. Let σ in $E_2(\lambda)$ be the epoch right after $E_4(\lambda)$. By construction, there is at most $K^{cl}\cdot(K+1)$ epochs in $E_4(\lambda)$ which is the size of the history H_λ^{cl}. Hence, at the beginning of σ, the history H_λ^{cl} contains all the labels the processor λ has produced during E_4 as well as all the K^{cl} (exactly) labels it has received during E_4. Since there is at most K^{cl} candidates label for canceling in the system, necessarily, in the first configuration of σ, the history H_λ^{cl} contains every candidates label for canceling present in the whole system. And since l_σ is greater, by construction, than every label in the history H_λ^{cl}, l_σ was not present in the entry λ of some tag in the configuration that precedes σ and it cannot be canceled by any other label present in the the system. In addition, by construction, E_2 only contains interrupts like $[\lambda, \infty]$ or $[\lambda, cl]$. From what we said about l_σ, the interrupt at the end of σ is necessarily $[\lambda, \infty]$. In other words, the epoch σ is a 0-safe epoch. □

Note that the epoch found in the proof is not necessarily the unique 0-safe epoch in $E_c(\lambda)$. The idea is only to prove that there exists a practically infinite epoch. If the first epoch σ at λ ends because the corresponding label l_σ in the entry μ_σ gets canceled, but lasts a practically infinite long time, then this epoch can be considered, from an informal point of view, safe. One could worry about having only very "short" epochs at λ due to some inconsistencies (canceling labels or overflows) in the system. Prop. 25 shows that every time a "short" epoch ends, the system somehow loses one of its inconsistencies, and, eventually, the proposer λ reaches a practically infinite epoch. Note also that a 0-safe epoch and a 1-safe or a 2-safe epoch are, in practice, as long as each other. Indeed, any h-safe epoch with h very small compared to $2^{\mathfrak{b}}$ can be considered practically infinite. Whether h can be considered very small depends on the concrete timescale of the system. Besides, every processor α always checks that the entry α is valid. Doing so the processor α still works to find a "winning" label for its entry α. In that case, if the entry μ becomes invalid, then the entry α is ready to be used, and a safe epoch can start without waiting any longer.

10.3 Safety

Definitions

To prove the safety property within an execution segment, we have to focus on the events that correspond to deciding a proposal, e.g., (v, t, p) at processor α (v being a tag, t a ballot integer, p a sequence of requests). Such an event may be due to corrupted messages in the communication channels an any stage of the Paxos algorithm. Indeed, a proposer computes the proposal it will send in its phase 2 thanks to the replies it has received at the end of its phase 1. Hence, if one of these messages is corrupted, then the safety might be violated. However, there is a finite number of corrupted messages since the capacity of the communication channels is finite. To formally deal with these issues, we define the notion of scenario that corresponds to specific chain of events involved in the Paxos algorithm. Consider an execution segment $E = (\gamma_k)_{k_0 \leq k \leq k_1}$. A

scenario in E is a sequence $U = (U_i)_{0 \leq i < I}$ where each U_i is a collection of events in E. In addition, every event in U_i happens before every event in U_{i+1}.

Definition 11 (Phase Scenario)**.** *Consider a proposer* ρ*, an acceptor* α*, quorums* S *and* Q *of acceptors, a tag* v*, a ballot integer* t*, and a sequence of requests* p*.*

A phase 1 scenario *is defined as follows. The proposer* ρ *broadcasts a message* p1a *containing the tag* v*, and ballot integer* t*. Every acceptor in the quorum* S *receives this message and adopts*[5] *the tag* v*. Every processor* α *in the quorum* S *replies to the proposer* ρ *a* p1b *message telling they adopted the couple* (v, t)*, and containing the last proposal they accepted. These messages are received by* ρ*. We denote this scenario by* $\rho \xrightarrow{p1a} (S, v, t) \xrightarrow{p1b} \rho$*.*

A phase 2 scenario with acceptation *is defined as follows. The proposer* ρ *broadcasts a* p2a *message containing the tag* v*, the ballot integer* t*, and the proposed sequence of requests* p*. The acceptor* α *accepts the proposal* (v, t, p)*. We denote this scenario by* $\rho \xrightarrow{p2a} (\alpha, v, t, p)$*.*

A phase 2 scenario with quorum acceptation *is defined as follows. The proposer* ρ *broadcasts a* p2a *message containing the tag* v*, the ballot integer* t*, and the proposed sequence of requests* p*. Every acceptor in the quorum* Q *accepts the proposal* (v, t, p)*. Every acceptor* α *in the quorum* Q *sends to the proposer* ρ *a* p2b *message telling that it has accepted the proposal* (v, t, p)*. The proposer* ρ *receives these messages. We denote this scenario by* $\rho \xrightarrow{p2a} (Q, v, t, p) \xrightarrow{p2b} \rho$*.*

A phase 2 scenario with decision *is defined as follows. The proposer* ρ *broadcasts a* p2a *message containing the tag* v*, the ballot integer* t*, and the proposed sequence of requests* p*. Every acceptor in the quorum* Q *accepts the proposal* (v, t, p)*. Every acceptor* α *in the quorum* Q *sends to the proposer* ρ *a* p2b *message telling that it has accepted the proposal* (v, t, p)*. The proposer* ρ *receives these messages. The proposer* ρ *sends a decision message containing the proposal* (v, t, p)*. The processor* α *receives this message, accepts and decides on the proposal* (v, t, p)*. We denote this scenario by* $\rho \xrightarrow{p2a} (Q, v, t, p) \xrightarrow{p2b} \rho \xrightarrow{dec} (\alpha, v, t, p)$*.*

In all the previous cases, we say that the phase scenarios are conducted by the proposer ρ *and* use the ballot (v, t)*.*

A simple acceptation scenario is simply a basic execution of the Paxos algorithm that leads a processor to either accept a proposal, or decide on a proposal (accepting it by the way).

Definition 12 (Simple Acceptation Scenario)**.** *A* simple acceptation scenario *is the concatenation of a phase 1 scenario, followed by a finite number of phase 2 scenarios with quorum acceptation, and ending with a phase* 2 *scenario with either acceptation, or decision; all the phase scenarios being conducted by the same proposer* ρ*, and using the same ballot* (v, t)*. Let* S *be the quorum of acceptors in the phase 1 scenario,* p *be the sequence of requests accepted (or decided on) in the last event of the scenario, and* α *be the corresponding acceptor. If the last phase scenario is a phase scenario with acceptation, then we denote the simple acceptation scenario by* $\rho \xrightarrow{p1a} (S, v, t) \rightsquigarrow \rho \xrightarrow{p2a} (\alpha, v, t, p)$*. If the last*

[5] Recall that this means the acceptor, say α, copies the entry $v[\chi(v)]$ in the entry $v_\alpha[\chi(v)]$.

phase scenario is a phase scenario with decision, then we denote the simple acceptation scenario by $\rho \xrightarrow{p1a} (S,v,t) \rightsquigarrow \rho \xrightarrow{p2a} (Q,v,t,p) \xrightarrow{p2b} \rho \xrightarrow{dec} (\alpha,v,t,p)$. *When we want to indicate that both cases are possible, we simply denote the simple acceptation scenario by* $(\rho,S,v,t) \rightsquigarrow (\alpha,v,t,p)$.

Since the initial configuration is arbitrary, there is necessarily a prefix of the execution during which the behaviour of the system is unknown. In particular, it may produce incoherent messages that can alter future events. We refer to these as *fake messages*.

Definition 13 (Fake Message)**.** *Given an execution segment* $E = (\gamma_k)_{k_0 \leq k \leq k_1}$, *a* fake message relative to E, *or simply a fake message, is a message that is in the communication channels in the first configuration* γ_{k_0} *in* E.

This definition of fake messages comprises the messages at the beginning of E that were not sent by any processor, but also messages produced in the prefix of execution that precedes E. We now define the analogues of phase scenarios when a fake message is involved.

Definition 14 (Fake Phase Scenario)**.** *Consider a proposer* ρ, *an acceptor* α, *quorums* S *and* Q *of acceptors, a tag* v, *a ballot integer* t, *and a sequence of requests* p. *Fix an execution segment* E. *A* fake phase scenario relative to E *is one of the following scenario.*

(Fake phase 1 scenario) *The proposer* ρ *sends a* p1a *message with ballot* (v,t). *It receives positive replies from a quorum* S, *one of these replies at least being fake (i.e. it was not actually sent by an acceptor). We denote this fake phase scenario by* $\rho \xrightarrow{p1a} (S,v,t) \xrightarrow{fake\ p1b} \rho$.

(Fake phase 2 scenario with acceptation) *The acceptor* α *receives a fake* p2a *with proposal* (v,t,p) *that seems to come from the processor* ρ. *The acceptor* α *accepts the proposal. We denote this scenario by* $\rho \xrightarrow{fake\ p2a} (\alpha,v,t,p)$.

(Fake phase 2 scenario with quorum acceptation) *The proposer* ρ *sends a* p2a *message with proposal* (v,t,p). *The proposer* ρ *receives positive replies from a quorum* Q, *one of these replies, at least, being fake. Then* ρ *sends a decision message with proposal* (v,t,p) *to the acceptor* α, *and* α *decides accordingly. We denote this scenario by* $\rho \xrightarrow{p2a} (Q,v,t,p) \xrightarrow{fake\ p2b} \rho \xrightarrow{dec} (\alpha,v,t,p)$.

(Fake phase 2 scenario with decision) *The acceptor* α *receives a fake decision message with proposal* (v,t,p) *which seems to come from the proposer* ρ. *The acceptor* α *decides accordingly. We denote this scenario by* $\rho \xrightarrow{fake\ dec} (\alpha,v,t,p)$.

Definition 15 (Simple Fake Acceptation Scenario)**.** *A* simple fake acceptation scenario *is either a fake phase 2 scenario with acceptation, a fake phase 2 scenario with quorum acceptation, a fake phase 2 scenario with decision, or the concatenation of a fake phase 1 scenario, followed by a finite number of (non-fake) phase 2 scenarios with quorum acceptation, and ending with a (non-fake) phase 2 scenario with either an acceptation, or a decision; all the scenarios being conducted by the same proposer* ρ, *and using the same ballot* (v,t). *We often denote this kind of scenarios by* $fake \rightsquigarrow (\alpha,v,t,p)$ *where* (α,v,t,p) *refers to the last acceptation (or decision) event.*

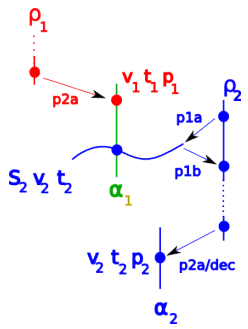


Figure 10.1: Composition of scenarios - Time flows downward, straight lines are local executions, arrows represent messages.

A simple fake acceptation scenario is somehow similar to a simple acceptation scenario except for the fact that at least one fake message (relative to the given execution segment) is involved during the scenario.

Definition 16 (Composition)**.** *Consider two simple scenarios*

$$U = X \rightsquigarrow (\alpha_1, v_1, t_1, p_1)$$
$$V = (\rho_2, S_2, v_2, t_2) \rightsquigarrow (\alpha_2, v_2, t_2, p_2)$$

where $X = fake$ *or* $X = (\rho_1, S_1, v_1, t_1)$ *such that the following three conditions are satisfied. (1) The processor* α_1 *belongs to* S_2 *(2) Let* e_2 *be the event that corresponds to* α_1 *sending a* p1b *message in scenario* V*. Then the event "*α_1 *accepts the proposal* (v_1, t_1, p_1)*" from* U *is the last acceptation event before* e_2 *occurring at* α_1*. In addition, the proposer* ρ_2 *selects the proposal* (t_1, p_1) *as the highest-numbered proposal at the end of the Paxos phase* 1*. In particular,* p_1 *is a prefix of* p_2*, i.e.,* $p_1 \sqsubseteq p_2$*. (3) All the tags involved share the same first valid entry, the same corresponding label.*

Then the composition *of the two simple scenarios is the concatenation the scenarios* U *and* V*. This scenario is denoted by* $X \rightsquigarrow (\alpha_1, v_1, t_1, p_1) \rightarrow (\rho_2, S_2, v_2, t_2) \rightsquigarrow (\alpha_2, v_2, t_2, p_2)$*. Note also that the ballot integer is strictly increasing along the simple scenarios.*

Definition 17 (Acceptation Scenario)**.** *Given an execution segment* E*, an* acceptation scenario *is the composition* U *of simple acceptation scenarios* $U_1, \ldots, U_r$ *where* U_1 *is either a simple acceptation scenario or a simple fake acceptation scenario relative to* E*, whereas the other are real (i.e. non-fake) simple acceptation scenarios. We denote it by* $X \rightsquigarrow (\alpha_1, v_1, t_1, p_1) \rightarrow (\rho_2, S_2, v_2, t_2) \rightsquigarrow (\alpha_2, v_2, t_2, p_1) \ldots (\rho_r, S_r, v_r, t_r) \rightsquigarrow (\alpha_r, v_r, t_r, p_r)$ *where* X *is either* $fake$ *or some* (ρ_1, S_1, v_1, t_1)*.*

An acceptation scenario whose first simple scenario is not fake relative to E *is called* real acceptation scenario relative to E*. An acceptation scenario whose first simple scenario is fake relative to* E *is called* fake acceptation scenario relative to E*.*

Given an acceptation event or a decision event, there is always at least one way to trace back the scenario that has lead to this event. If one of these

scenarios involve a fake message, then we cannot control the safety property. Besides, all the tags involved share the same first valid entry μ and the same corresponding label l. Also, the ballot integer value, as well as the sequence of requests, is increasing along the acceptation scenario; i.e., if $i < j$, then $t_i < t_j$ and $p_i \sqsubseteq p_j$.

Definition 18 (Real event)**.** *Consider an event e that corresponds to some processor accepting a proposal, let U be the simple acceptation scenarios that ends with the event e. The event e is said to be* real relative to an execution segment E *if U is a real simple acceptation scenario relative to E. The event e is said to be* fake relative to E *otherwise.*

Definition 19 (Simple Scenario Characteristic)**.** *The* characteristic *of a simple acceptation scenario U with tag v, ballot integer t, is the tuple $char(U) = (\chi(v), v[\chi(v)].l, t)$.*

When a proposer λ manages to reach a h-safe epoch (with low h), then λ cannot see any event that would cause an interrupt during its epoch. This remark allows to associate to such an epoch at λ, a globally defined period of time, namely the *zone observed by* λ. In Prop. 27, we show that during this period of time, under some specific assumptions, the safety property is ensured.

Definition 20 (Observed Zone)**.** *Consider an execution E. Let λ be a proposer and let Σ be an execution segment such that the local execution $\sigma = \Sigma(\lambda)$ at λ is a h-safe epoch. We denote by F the suffix of the execution that starts with Σ. Assume that λ hears from at least two quorums during its epoch σ. Let Q^0, Q^f be the first and last quorums respectively whose messages are processed by the proposer λ during σ. For each processor α in Q^0 (resp. Q^f), we denote by $e^0(\alpha)$ (resp. $e^f(\alpha)$) the event that corresponds to α sending to λ a message received in the phase that corresponds to Q^0 (resp. Q^f).*

The zone observed by λ during the epoch σ, *namely $Z(F, \lambda, \sigma)$, is the set of acceptation scenarios relative to F described as follows. An acceptation scenario relative to F belongs to $Z(F, \lambda, \sigma)$ if and only if it ends with a real acceptation (or decision) event (relative to F) that does not happen after the end of σ and it contains a real simple acceptation scenario $U = (\rho, S, v, t) \rightsquigarrow (\beta, v, t, p)$ such that there exists an acceptor α in $S \cap Q^0 \cap Q^f$ at which the event $e^0(\alpha)$ happens before the event e that corresponds to sending a p1b message in U, and the event e happens before the event $e^f(\alpha)$ (cf. Figure 10.2).*

Results

The following lemma highlights the causal relation between an epoch at some proposer, and a local execution at another processor which undergoes a cycle of labels.

Lemma 25 (Epoch and Cycle of Labels)**.** *Consider an execution E. Let λ be a processor and consider an execution segment Σ such that the local execution $\sigma = \Sigma(\lambda)$ is an epoch at λ. We denote by F the suffix of the execution E that starts with Σ. Consider a processor ρ and a finite execution segment G in F as follows: G starts in Σ and induces a local execution $G(\rho)$ at ρ such that it starts and ends with the first valid entry of the tag v_ρ being equal to μ_σ*

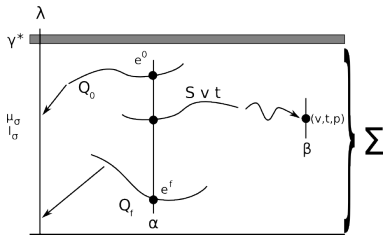


Figure 10.2: Scenario $(\rho, S, v, t) \rightsquigarrow (\beta, v, t, p)$ in $Z(F, \lambda, \sigma)$ - Time flows downward, straight lines are local executions, curves are send/receive events, arrows represent messages.

and containing the label l_σ, and the label field in the entry $v_\rho[\mu_\sigma]$ undergoes a cycle of labels during $G(\rho)$. Assume that, if $\mu_\sigma < \lambda$, the processor μ_σ does not produce any label during G. Then $\mu_\sigma = \lambda$ and the last event of σ happens before the last event of $G(\rho)$.

Proof. By Lem. 23, since the entry $v_\rho[\lambda]$ remains valid after the re-adoption of the label l at the end of $G(\rho)$, the proposer μ_σ must have produced some label l' during G (hence $\mu_\sigma = \lambda$) that was received by ρ during G. Necessarily, the production of l' happens after the last event of σ at λ, thus the last event of $G(\rho)$ at ρ also happens after the last event of σ at λ. □

We now focus on proving the safety property. The following crucial proposition focuses on real simple acceptation scenarios.

Proposition 26 (Safety - Weak Version)**.** *Consider an execution E. Let λ be a processor and let Σ be an execution segment such that the local execution $\sigma = \Sigma(\lambda)$ at λ is an h-safe epoch. We denote by F the suffix of the execution that starts with Σ. Consider the two simple scenarios $U_1 = \rho_1 \xrightarrow{p1a} (S_1, v_1, t_1) \rightsquigarrow \rho_1 \xrightarrow{p2a} (Q_1, v_1, t_1, p_1) \xrightarrow{p2b} \rho_1 \xrightarrow{dec} (\alpha_1, v_1, t_1, p_1)$ and $U_2 = (\rho_2, S_2, v_2, t_2) \rightsquigarrow (\alpha_2, v_2, t_2, p_2)$ with characteristics $(\mu_\sigma, l_\sigma, t_1)$ and $(\mu_\sigma, l_\sigma, t_2)$ respectively.*

We denote by e_i the acceptation event $(\alpha_i, v_i, t_i, p_i)$. Assume that the events e_1 and e_2 occur in F and that $h \leq t_1 \leq t_2$. In addition, assume that, if $\mu_\sigma < \lambda$, then the processor μ_σ does not produce any label during F. We then have two cases: (a) If $t_1 = t_2$, then either $p_1 \sqsubseteq p_2$, or $p_2 \sqsubseteq p_1$, or the last event of σ happens before one of the event e_1 or e_2. (b) If $t_1 < t_2$, then $p_1 \sqsubseteq p_2$ or the last event of σ happens before one of the event e_1 or e_2.

Proof. We assume that both events e_1 and e_2 do not happen after the last event of σ and we prove the result. We denote by γ^* the configuration right before Σ. We prove the result by induction on the value of t_2.

(Bootstrapping). We first assume that $t_2 = t_1$. Recall the ballot integers include the identifiers of the proposer, hence $\rho_1 = \rho_2$. If $p_1 \not\sqsubseteq p_2$ and $p_2 \not\sqsubseteq p_1$, then ρ_1 has sent two $p2a$ messages with different proposals and the same ballot. Let e and f be the events corresponding to these two sendings. None of the events e and f occurs in the execution prefix A, otherwise, since e_1 and e_2 occur in F, the configuration γ^* would contain a ballot (x, t) with $x[\mu_\sigma].l = l_\sigma$ and $t \geq h$; this is a contradiction since σ is h-safe.

Hence, e and f occur in F. The fact that $p_1 \not\sqsubseteq p_2$ and $p_2 \not\sqsubseteq p_1$ implies that there must be a cycle of labels in the entry $v_{\rho_1}[\mu_\sigma]$ between the e and f. By Lem. 25, this implies that the last event of σ happens before the event e_1 or e_2; this is a contradiction. Hence, $p_1 \sqsubseteq p_2$ or $p_2 \sqsubseteq p_1$.

(Induction). Now, $t_1 < t_2$ and we assume the result holds for every value t such that $t_1 \leq t < t_2$. Pick some acceptor β in $Q_1 \cap S_2$. From its point of view, there are two events f_1 and f_2 at β that respectively correspond to the acceptation of the proposal (v_1, t_1, p_1) in the scenario U_1 (reception of a $p2a$ message), and the adoption of the ballot (v_2, t_2) in the scenario U_2 (reception of a $p1a$ message). First, the events f_1 and f_2 do not occur in the execution prefix A. Otherwise there would exist a ballot (x, t) in γ^* such that $x[\mu_\sigma].l = l_\sigma$ and $t \geq h$; this is a contradiction, since σ is h-safe. Hence, f_1 and f_2 occur in the suffix F.

We claim that f_1 happens before f_2. Otherwise, since $t_2 > t_1$, there must be a cycle of labels in the field $v_\beta[\mu_\sigma].l$. By Lem. 25, this implies that the last event of σ happens before the event f_1, and thus before the event e_1; contradiction. Hence, f_1 happens before f_2.

We claim that the $p1b$ message the acceptor β has sent contains a non-null lastly accepted proposal (t, p) such that $t_1 \leq t < t_2$ and $p_1 \sqsubseteq p$. Otherwise, there is a cycle of labels in the field $v_\beta[\mu_\sigma].l$, which implies that the last event of σ happens before the event f_2, and thus before the event e_2 also; this is impossible.

Now, the proposer ρ_2 receives a set of proposals from the acceptors of the quorum S_2, including at least one non-null proposal from β. Then, it selects among the replies, the accepted proposal (t_c, p_c) with the highest ballot integer, and highest request sequence length (lexicographical order). Since ρ_2 has received the proposal (t, p) from β, we then have $h \leq t_1 \leq t \leq t_c < t_2$ and $(t, |p|) \leq (t_c, |p_c|)$ (lexicographically).

Let β_c be the proposer in S_2 which has sent to ρ_2 the proposal (t_c, p_c) in the $p1b$ message. There is an event f_c in F that corresponds to β_c accepting the proposal (t_c, p_c). Otherwise there would exist a ballot (x, t') in γ^* such that $x[\mu_\sigma].l = l_\sigma$ and $t' \geq h$; this is a contradiction, since σ is h-safe.

Consider the simple acceptation scenario V_c that ends with f_c, and let $char(V_c) = (\mu_c, l_c, t_c)$ be its characteristic. Since f_c is the last acceptation event before β_c replies to ρ_2 (with a $p1a$ message), we must have $(\mu_c, l_c) = (\mu_\sigma, l_\sigma)$; otherwise, the accepted variable $accepted_{\beta_c}$ would have been cleared (epoch change at β_c), and β_c would have not sent the non-null proposal (t_c, p_c) to ρ_2. If V_c were a fake simple acceptation scenario, then there would exist a ballot (x, t') in γ^* such that $x[\mu_\sigma].l = l_\sigma$ and $t' \geq h$; this is impossible, since σ is h-safe. Hence V_c is a real simple acceptation scenario.

By applying the induction hypothesis to V_c, and since f_c cannot happen after the last event of σ (otherwise e_2 would also happen after it), we have two cases. The case (A) $t_1 = t_c$. Then $p_1 \sqsubseteq p_c$ or $p_c \sqsubseteq p_1$. But, the fact that $(t, |p|) \leq (t_c, |p_c|)$ (lexicographically) and $p_1 \sqsubseteq p$ implies that $|p_c| \geq |p| \geq |p_1|$, and thus $p_1 \sqsubseteq p_c$. The case (B) $t_1 < t_c$. But then $p_1 \sqsubseteq p_c$.

In all cases, we have $p_1 \sqsubseteq p_c$. But, we also have $p_c \sqsubseteq p_2$ (scenario U_2), hence $p_1 \sqsubseteq p_2$. □

We get the following corollary for decision events.

Corollary 2. *Consider an execution E. Let λ be a processor and let Σ be an execution segment such that the local execution $\sigma = \Sigma(\lambda)$ at λ is an h-safe epoch. We denote by F the suffix of the execution that starts with Σ.*

Consider two decision events $e_i = (\alpha_i, v_i, t_i, p_i)$, $i = 1, 2$, such that $\chi(v_i) = \mu_\sigma$, $v_i[\mu_\sigma].l = l_\sigma$ and $t_i \geq h$. Assume that both events e_1 and e_2 are real decision events relative to F. In addition, assume that, if $\mu_\sigma < \lambda$, then the processor μ_σ does not produce any label during F. Then either $p_1 \sqsubseteq p_2$, $p_2 \sqsubseteq p_1$ or the last event of σ happens before one of the event e_1 or e_2.

Proof. Since e_1 and e_2 are real decision events relative to F, there are two real simple acceptation scenarios with decision U_1 and U_2 ending with e_1 and e_2. Let's denote them as follows:

$$U_1 = \rho_1 \xrightarrow{p1a} (S_1, v_1, t_1) \rightsquigarrow \rho_1 \xrightarrow{p2a} (Q_1, v_1, t_1, p_1) \xrightarrow{p2b} \rho_1 \xrightarrow{dec} (\beta_1, v_1, t_1, p_1) \tag{10.2}$$

$$U_2 = \rho_2 \xrightarrow{p1a} (S_2, v_2, t_2) \rightsquigarrow \rho_2 \xrightarrow{p2a} (Q_2, v_2, t_2, p_2) \xrightarrow{p2b} \rho_2 \xrightarrow{dec} (\beta_2, v_2, t_2, p_2) \tag{10.3}$$

They have characteristics $(\mu_\sigma, l_\sigma, t_1)$ and $(\mu_\sigma, l_\sigma, t_2)$ respectively and $t_1, t_2 \geq h$. Whether $t_1 \leq t_2$ or $t_2 \leq t_1$, Prop. 26 yields the result. □

Finally, we can now state the main proposition of this section: within the observed zone associated the h-safe epoch at some proposer, the safety property is ensured.

Proposition 27 (Safety)**.** *Consider an execution E, a proposer λ proposer and an execution segment Σ such that the local execution $\sigma = \Sigma(\lambda)$ at λ is a h-safe epoch for some bounded integer h. We denote by F the suffix of execution that starts with Σ. Assume that the observed zone $Z(F, \lambda, \sigma)$ is defined and that, if $\mu_\sigma < \lambda$, then the processor μ_σ does not produce any label during F. Consider two scenarios U_1 and U_2 in $Z(F, \lambda, \sigma)$ ending with acceptation events $e_1 = (\alpha_1, v_1, t_1, p_1)$ and $e_2 = (\alpha_2, v_2, t_2, p_2)$. Let $\mu_i = \chi(v_i)$ and $l_i = v_i[\mu_i]$, $i = 1, 2$, and assume that $\mu_\sigma \leq \min(\mu_1, \mu_2)$ and $t_1, t_2 \geq h$. Then $(\mu_1, l_1) = (\mu_2, l_2) = (\mu_\sigma, l_\sigma)$, and $p_1 \sqsubseteq p_2$ or $p_2 \sqsubseteq p_1$.*

Proof. Assume that $\mu_1 > \mu_\sigma$. By definition of the observed zone $Z(F, \lambda, \sigma)$, there exists a simple acceptation scenario $V = (\rho, S, v, t) \rightsquigarrow (\beta, v, t, p)$ in U_1 and an acceptor α in $S \cap Q^0 \cap Q^f$ such that we have the happen-before relations $e^0(\alpha) \rightsquigarrow e \rightsquigarrow e^f(\alpha)$, where e is the event that corresponds to α sending a $p1b$ message in the scenario V. We also have $\chi(v) = \mu_1$ and $v[\mu_1].l = l_1$.

At $e^0(\alpha)$ and $e^f(\alpha)$, messages are sent to λ and are processed during σ. Hence, the corresponding tag values of the variable v_α must use the entry μ_σ and the label l_σ. Otherwise, the message either is not processed or causes an interrupt at processor λ.

Now, at event e, the first valid entry of the variable v_α is $\mu_1 > \mu_\sigma$ which implies that the entry μ_σ is invalid.

Hence, between $e^0(\alpha)$ and $e^f(\alpha)$, the entry $v_\alpha[\mu_\sigma]$ becomes invalid and valid again. Thus, there has been a cycle of labels in the label field $v_\alpha[\lambda].l$. Lem. 25 implies that the last event of σ happens before $e^f(\alpha)$; by the definition of $e^f(\alpha)$, this is a contradiction. Therefore $\mu_1 = \mu_\sigma$.

If $l_1 \neq l_\sigma$, then there is also a cycle of labels in the entry $v_\alpha[\mu_\sigma]$ between $e^0(\alpha)$ and $e^f(\alpha)$, which leads to a contradiction again, thanks to the same argument. Therefore, $l_1 = l_\sigma$.

Of course, the previous argument also applies to U_2 and shows that $(\mu_2, l_2) = (\mu_\sigma, l_\sigma)$. Therefore, Corollary 2, the fact that $t_1, t_2 \geq h$ and the fact that the two acceptation events e_1, e_2 do not happen after the end of σ imply that $p_1 \sqsubseteq p_2$ or $p_2 \sqsubseteq p_1$. □

In the case $\mu_\sigma < \lambda$, assuming that μ_σ does not produce any label during F means that the proposer λ should be the live processor with the lowest identifier. To deal with this issue, one can use a failure detector.

10.4 Liveness

Liveness in Paxos is not guaranteed unless there is a unique proposer. The original Paxos algorithm assumes that the choice of a distinguished proposer is done through an external module. In the sequel, we present an implementation of a self-stabilizing failure detector that works under a partial synchrony assumption. Note that this assumption is strong enough to implement an eventual perfect failure detector, but such a failure detector is not mandatory for our tag system to stabilize. This brief section simply explains how a *self-stabilizing* implementation can be done; which is, although not difficult, not obvious either. Each processor α has a vector L_α indexed by the processor identifiers; each entry $L_\alpha[\mu]$ is an integer whose value is comprised between 0 and some predefined maximum constant W. Every processor α keeps broadcasting a heartbeat message $\langle hb, \alpha \rangle$ containing its identifier (e.g., by using [42, 43]). When the processor α receives a heartbeat from processor β, it sets the entry $L_\alpha[\beta]$ to zero, and increments the value of every entry $L_\alpha[\rho]$, $\rho \neq \beta$ that has value less than W. The detector output at processor α is the list F_α of every identifier μ such that $L_\alpha[\mu] = W$. In other words, the processor α assesses that the processor β has crashed if and only if $L_\alpha[\beta] = W$.

(Interleaving of Heartbeats). For any two live processors α and β, between two receptions of heartbeat $\langle hb, \beta \rangle$ at processor α, there are strictly less than W receptions of heartbeats from other processors. Under this condition, for every processor α, if the processor β is alive, then eventually the identifier β does not belong to the list F_α. A distinguished proposer ρ can be defined as follows: $\rho = \min(\mu;\ L_\rho[\mu] < W)$.

Part III

Conclusion

Chapter 11

Perspectives

As explained in the introduction of this thesis, there are two approaches to solve a problem in a given model. The explicit approach consists in having explicit assumptions on the parameters of the model, and try to use them to solve the problem. The other approach, namely the implicit approach, takes the opposite point of view: one augments the system with a (distributed) oracle strong enough to solve the problem, and looks for the minimal oracle able to solve this problem. Our results naturally follow this distinction. In the following, we sum up our results, and highlight interesting perspectives for future work.

11.1 Population Protocols

Explicit Approach - Fairness

Fairness and Solvability. In Chap. 6, we have studied the leader election problem. In particular, we have shown that, if the agents are uniformly initialized, then the problem is impossible to solve using the local fairness, whereas the problem has a solution when considering the global fairness. The reason for the impossibility result is that there are locally fair schedules which maintain a form of "symmetry" in the population(see the notion of graph coverings in Chap. 6, Sec. 6.2). On the other hand, the global fairness mimics a form of randomness: it basically ensures that any configuration reachable infinitely often is actually reached infinitely often. It is folklore that randomization is useful to break symmetry, and this somehow explains why the global fairness is sufficient to solve the leader election problem. An interesting perspective is to look for intermediate fairness conditions between the local fairness and global fairness and see where exactly the impossibility/possibility barrier is located.

Graph family $\mathcal{F}$	Initialization	Fairness	Notes
Contains a covering	Uniform	Local	LE is impossible
Arbitrary	Uniform	Global	LE is possible

Fairness and Efficiency. In Chap. 4, we have studied the (adapted) ZebraNet protocol as well as two variants. By using an explicit assumption on the schedules under the form of cover times, we were able to analytically derive

tight bounds for the worst-case convergence time. The same techniques can be used to provide a qualitative analysis given a particular distribution of cover times (see Chap. 4, Sec. 4.6).

A natural extension of this work would be to perform an *average-case* analysis. However, this task is not easy to achieve. A usual average analysis consists in enumerating the fair schedules that induce a given convergence time, but, doing so, one is necessarily confronted with a combinatorial explosion.

Another approach is to take a probabilistic scheduler: at each step, an edge is randomly selected according to some probability distribution on the edges of the communication graph. The notion of speed of agents can then be modeled by a non-uniform distribution. The whole system can be seen as a Markov process; the Markov state corresponding to a configuration, and the Markov transition matrix is computed from the probability distribution on the communication graph's edges. Doing so, one is again confronted with a combinatorial explosion since the number of configurations is exponential in the number of agents. But, there is chance that the tools developed in the domain of Markov processes could be useful in this analysis.

Implicit Approach - Oracles

The implicit approach based on oracles is relatively new in the population protocols model. As far as we know, oracles adapted to population protocols are presented in [65, 46], but no work has focused on comparing oracles, and looking for minimal ones. In this thesis, we have paved the way to a general oracle-based approach in the model of population protocols. We sum up our results in increasing order of the power of the introduced oracles.

Consensus. In Chap. 5, we have studied the consensus problem and a variant, called the symmetric consensus, which guarantees that the decision value is independent of the distribution of the initial values among the agents. We have proven that the consensus is impossible without oracles. We have then introduced a class of oracles, called $Mnemosyne$, which mainly notifies each agent in the population about the presence of specific patterns in their causal pasts. This class of oracles is a natural adaptation of the classical failure detectors [32] to the population protocol model. They observe the schedule of meetings, output binary values, are anonymous, and are not required to give the correct information right on time. We have exhibited the oracle $DejaVu$ from this class, which, basically, notifies an agent when it has indirectly seen every other agent. We have then proven that $DejaVu$ is the weakest oracle in $Mnemosyne$ for solving the symmetric consensus problem.

An open problem is to find the weakest oracle in $Mnemosyne$ for solving the original consensus problem. An intuitive candidate is a leader-like oracle: in each execution, a unique agent is eventually notified by the oracle. With such an oracle, the selected agent may decide on its own input value and force the others to decide accordingly. Note that this leader-like oracle may notifies an agent which has not yet seen (indirectly) every other agent. Hence, requiring this agent to be unique is crucial, as otherwise, two agents could decide on different proposals.

Note however that finding the weakest oracle in $Mnemosyne$ for consensus is a difficult task. Indeed, our proof for the symmetric consensus relies on the

construction of a contradictory execution, for which the symmetry condition is crucial. To adapt the proof, one needs a more detailed analysis of the behaviour associated with an unknown population protocol (see the remarks below).

Self-Stabilizing Leader Election. In the second part of Chap. 6, we have studied the self-stabilizing leader election (SSLE) problem. It turns out that this problem is impossible in most cases [10]. Following the implicit approach, the oracle $\Omega?$ has been introduced in [46] for solving the problem over complete graphs and rings. A large part of our work has been devoted to extend these results as shown in the table below[1]. We have proven that $\Omega?$ is strong enough to solve SSLE over graphs with bounded-degree. We then introduced a lattice of oracles $\{\Omega?(d)^{\otimes k}\}_{d,k\geq 1}$ that generalize the Fischer and Jiang's oracle $\Omega?$, and we have proven that $\Omega?(2)$ and $\Omega?^{\otimes 2}$ are sufficient to solve SSLE over arbitrary graphs.

Graph family $\mathcal{F}$	Initialization	Fairness	Notes
Bounded-degree	Arbitrary	Global	$SSLE \preccurlyeq \Omega?$
Arbitrary	Arbitrary	Global	$SSLE \preccurlyeq \Omega?(2)$
			$SSLE \preccurlyeq \Omega?^{\otimes 2}$
Rings	Arbitrary	Global	$SSLE \simeq \Omega?$
Non-simple family	Arbitrary	Global	$SSLE^{\otimes k} \not\preccurlyeq \Omega?$

We now turn to the main motivation of the implicit approach: finding minimal oracles. In this direction, we have proven that the problem SSLE and the oracle $\Omega?$, seen as a problem too, are equivalent over rings. In particular, this implies that any oracle strong enough to solve SSLE over rings can be used to build a self-stabilizing implementation of $\Omega?$ over rings. A simple generalization of the corresponding proof shows that, over rings, all the oracles in the lattice $\{\Omega?(d)^{\otimes k}\}_{d,k\geq 1}$ are equivalent (to SSLE); the lattice structure collapses over the rings.

It turns out that this phenomenon is related to the ring family. Indeed, we have shown that over a non-simple graph family, the problem of self-stabilizing implementation of $\Omega?$ is not weaker than, nor equivalent to the SSLE problem. Yet, although $\Omega?(2)$ and $\Omega?^{\otimes 2}$ are (strictly) stronger than SSLE over the arbitrary graph family, it is unknown whether $\Omega?$ is also stronger than SSLE. Besides, this suggests that the relations in the lattice $\{\Omega?(d)^{\otimes k}\}_{d,k\geq 1}$ become strict over a non-simple graph family. The main difficulty in proving the impossibility of a reduction between oracles like $\Omega?$ relies on the fact that these oracles can be composed with protocols using the feedback operation. This suggests investigating more closely the behaviour associated with population protocols.

Remarks. For both leader election, and consensus, the proofs of most of our results rely on the analysis of an unknown protocol. For instance, to prove that an oracle solving a problem is stronger than another one, we usually start from the existence of protocol that solves the problem given the first oracle. The difficulty lies in the fact that we do not have much information, a priori,

[1] In this conclusion, we denote both the behaviour $\mathcal{ELE}$ and the informal self-stabilizing leader election problem by SSLE.

on this protocol, besides the fact that it solves the given problem. However, we also know that it is a population protocol. And the behaviour associated with a population protocol is not completely arbitrary. The main obstacle to the extensions of our results, as stated above, is the lack of a more precise understanding of the behaviour associated with a population protocol.

Let's take a basic example coming from the leader election problem. One of the main difficulty is that a leader must try to "kill" the other leaders without killing itself. Most of our techniques are based on token circulation, and since the schedules are non-deterministic, the circulating tokens perform a sort of random walk. If no protection mechanism is established, this randomness may force a leader to kill itself. Yet, this randomness is the price to pay for using circulating tokens to communicate. Now, given an unknown protocol, we do not know how information is transmitted between the leaders. If we knew that it uses a form of token circulation, then many proofs, especially impossibility proofs, would be easier. Thus, an interesting perspective is to aim at a better understanding of the behaviour associated with a protocol; this should highlight, for instance, the constraints on how information is transmitted among the agents. This naturally leads to the following section.

The Model

To encompass the various kinds of oracles, we have developed a formal framework in Chap. 3. Our model is two-fold. On the first hand, there is a local description under the form of a population protocol (the list of rules). On the other hand, there is a global description under the form of a behaviour. Naturally, any local description (population protocol), associated with a context (graph family, fairness condition, etc.), yields a global description (behaviour). On both levels, we have notions of composition (parallel, serial, feedback) which are compatible (see structure theorems in Chap. 3, Sec. 3.3).

The notion of behaviour is general enough to model both oracles and problems. Put another way, oracles and problems live on the same level. Implementing an oracle (resp. solving a problem) amounts to design a protocol whose associated behaviour is a sub-behaviour of the oracle (resp. the problem). Solving a problem P using an oracle O amounts to design a protocol such that some composition involving the protocol's behaviour and the behaviour O yields a sub-behaviour of P. These natural definitions give a sound notion of comparison between behaviours (oracles, problems): the behaviour B_1 is weaker than or equivalent to the behaviour B_2 if there exists an implementation of B_1 using B_2. The induced comparison relation is a partial order on behaviours.

Therefore, this framework gives the basic settings for studying reductions between problems in the context of population protocols. However, as seen in the cases of consensus and leader election, an important line of research is to study the function that maps a protocol to its associated behaviour. With this objective in mind, a reformulation of the model in terms of category[2] theory should give interesting insights. Very briefly, a population protocol $\mathcal{A}$ with input set X and output set Y can be seen as a morphism $X \xrightarrow{\mathcal{A}} Y$ in some category PP. The morphisms can be composed using the serial composition. The

[2]More precisely, in terms of bicategories.

parallel and feedback yield a sort of traced monoidal structure. The category PP somehow represents the local or microscopic objects since a population protocol only describes how the states of two meeting agents are updated. On the other hand, we can also define a category Bhv of behaviours. A behaviour B with input set X and output set Y are the morphisms of the category. Composition of morphisms is given by the serial composition, and the parallel and feedback compositions yield a traced monoidal structure. The category Bhv represents the global or macroscopic objects, like oracles and problems. Given a context $\mathcal{C}$, thanks to the structure theorems (Chap. 3, Sec. 3.3, Th. 2), the behaviour map $Beh : \mathcal{A} \mapsto Beh(\mathcal{A}, \mathbb{C})$ can be seen as a functor from the category PP of local objects to the category Bhv of global objects. Such a reformulation may highlight analogies with other fields of research. For instance, a closely related formulation in automata theory has yielded a very interesting result: the functor behaviour arises from an adjunction [48]; this somehow characterizes the behaviours which are associated with automata. Adopting the same approach in our situation is an interesting perspective.

11.2 State-Machine Replication

Explicit approach - Enhancing the Algorithm

In the second part of this thesis, we have studied the problem of state-machine replication in the classical asynchronous message-passing model with crash failures and transient faults. Paxos [56, 57] is a well-known algorithmic scheme for implementing a replicated state-machine in the asynchronous message-passing model with crash failures, but it does not cope with transient faults. By replacing a core component of Paxos, namely the timestamp management, we have managed to design a *practically self-stabilizing* state-machine replication protocol. The algorithm ensures that, after the last transient fault, which set the replicas in an arbitrary configuration, eventually the replicas will reach a segment of execution, whose length[3] is large enough relatively to some predefined timescale, during which they behave as in the original Paxos algorithm.

A perspective in the short-term is to optimize our algorithm. For instance, the size of the label histories has been defined so as to correspond to the maximum number of *different* labels in the system. This maximum is, a priori, equal to the total label capacity of the system, but it is very unlikely that the system would hold so many different label values. In practice, one could tune the size of the label histories to correspond to an estimate of the number of different labels currently present. Another room for optimization is the type of value on which the replicas agree. In our work, for sake of clarity, the replicas agree on a growing sequence of requests of length less than a predefined maximum value. Doing so, if a replica is corrupted, it can rebuild a correct state by replaying the first requests. However, one could also reduce a prefix of the request sequence to the corresponding state: the corrupted replica would just need to access this state before executing the next requests. Moreover, if one can ensure that at some point the replicas are coherent, then, from this point on, the proposers could just propose the most recent requests, as in the original Paxos, instead of sending the whole sequence.

[3] Measured by the longest causal chain in the execution segment.

Another perspective would be to extend our work to tolerate byzantine failures. The Castro-Liskov algorithm [28], and the closely related Byzantine Paxos [60], are state-machine replication algorithms tolerating byzantine faults, but they require to be started in a correct initial configuration. Both algorithms rely on the use of "proofs" to guarantee the correctness of the messages. Adapting such proof system to the self-stabilizing case is an open challenging problem.

Implicit Approach - Conditions for Solvability

In the study of state-machine replication, our first goal was to design a practically self-stabilizing variant of Paxos. However, it is an open problem whether a self-stabilizing implementation of a replicated state-machine exists *in the strong sense*, i.e., an implementation that ensures that some infinite suffix (instead of a practically infinite segment) of the execution is correct. Or more precisely, it is an open problem to find the minimal conditions under which the self-stabilizing state-machine replication problem is solvable. This task is difficult. We have seen that the main obstacle to self-stabilization is due to the finite memory of the replicas, and the asynchrony of message-passing. Indeed, the memory finiteness implies that the replicas must somehow "forget" a part of their past, and, since the messages take an arbitrary amount of time to be delivered, the replicas may not distinguish old from new messages.

By slightly weakening the problem, i.e., by aiming at a *practically* self-stabilizing state-machine replication protocol, we were able to derive a solution by adapting Paxos. A possible line of attack for the problem stated above is to study the minimal conditions under which our algorithm implements a practically self-stabilizing replicated state-machine. For instance, we have seen that no particular condition is required for our tag system to stabilize, specific conditions (e.g. unique proposer) are required only during the stabilized period to ensure liveness.

Following this line of research, an interesting perspective is to translate our algorithm in the HO model, or a variant of it, introduced in [34]. In this work, the authors study the conditions under which the one-shot version of the original Paxos is able to solve the consensus problem. To do so, they introduce a model that abstracts from the inner details of the communications, and focuses on the effects of these communications. More precisely, each execution is a sequence of rounds, and during each round, each processor receives a set of messages, perform some local updates, and send new messages. A predicate over the sequence of sets of received messages encodes the effects of communication. Thanks to this implicit approach, the authors have managed to provide a clear picture of the conditions under which the one-shot version of Paxos solves the consensus problem. Adapting our algorithm to this model, or a variant, should provide interesting insights on the problem of (practical) self-stabilizing state-machine replication.

Bibliography

[1] The Dartmouth wireless trace archive - http://crawdad.cs.dartmouth.edu/. Dartmouth College, 2007.

[2] U. Abraham. Self-stabilizing timestamps. *Theor. Comput. Sci.*, 308(1-3):449–515, Nov. 2003.

[3] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theor. Comput. Sci.*, 220(1):3–30, 1999.

[4] N. Alon, H. Attiya, S. Dolev, S. Dubois, M. Potop-Butucaru, and S. Tixeuil. Pragmatic self-stabilization of atomic memory in message-passing systems. In *SSS*, pages 19–31, 2011.

[5] D. Angluin. Local and global properties in networks of processors. In *12th Symposium on the Theory of Computing*, pages 82–93. ACM, 1980.

[6] D. Angluin, J. Aspnes, M. Chan, H. Jiang, M. Fischer, and R. Peralta. Stably computable properties of network graphs. In *DCOSS*, pages 63–74. LNCS 3560, 2005.

[7] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.

[8] D. Angluin, J. Aspnes, and D. Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.

[9] D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.

[10] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing population protocols. *ACM Trans. Auton. Adapt. Syst.*, 3(4), 2008.

[11] H. Attiya and J. Welch. *Distributed Computing*. McGraw-Hill, 1998.

[12] J. Beauquier, P. Blanchard, and J. Burman. Self-stabilizing leader election in population protocols over arbitrary communication graphs. In *OPODIS*, pages 38–52, 2013.

[13] J. Beauquier, P. Blanchard, J. Burman, and S. Delaët. Tight complexity analysis of population protocols with cover times - the zebranet example. *Theor. Comput. Sci.* In Press, Corrected Proof. Available online 31 October 2012.

[14] J. Beauquier and J. Burman. Self-stabilizing synchronization in mobile sensor networks with covering. In *Distributed Computing in Sensor Systems, 6th IEEE International Conference, DCOSS 2010, Santa Barbara, CA, USA, June 21-23, 2010. Proceedings*, volume 6131 of *Lecture Notes in Computer Science*, pages 362–378. Springer, 2010.

[15] J. Beauquier and J. Burman. Self-stabilizing mutual exclusion and group mutual exclusion for population protocols with covering. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, volume 7109 of *Lecture Notes in Computer Science*, pages 235–250. Springer, 2011.

[16] J. Beauquier, J. Burman, J. Clement, and S. Kutten. On utilizing speed in networks of mobile agents. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 305–314. ACM, 2010.

[17] J. Beauquier, J. Burman, and S. Kutten. A self-stabilizing transformer for population protocols with covering. *Theor. Comput. Sci.*, 412(33):4247–4259, 2011.

[18] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.

[19] P. Blanchard, S. Dolev, J. Beauquier, and S. Delaët. Self-stabilizing paxos. *CoRR*, abs/1305.4263, 2013.

[20] P. Boldi, S. Shammah, S. Vigna, B. Codenotti, P. Gemmell, and J. Simon. Symmetry breaking in anonymous networks: Characterizations. In *ISTCS*, pages 16–26, 1996.

[21] F. Bonnet and M. Raynal. The price of anonymity: Optimal consensus despite asynchrony, crash and anonymity. In *DISC*, pages 341–355, 2009.

[22] F. Bonnet and M. Raynal. Anonymous asynchronous systems: The case of failure detectors. In *DISC*, pages 206–220, 2010.

[23] Z. Bouzid and C. Travers. Anonymity, Failures, Detectors and Consensus. Technical report, 2012.

[24] Z. Bouzid and C. Travers. Brief announcement: Anonymity, failures, detectors and consensus. In *DISC*, pages 427–428, 2012.

[25] H. Cai and D. Y. Eun. Crossing over the bounded domain: from exponential to power-law inter-meeting time in MANET. In *MOBICOM*, pages 159–170, 2007.

[26] S. Cai, T. Izumi, and K. Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory Comput. Syst.*, 50(3):433–445, 2012.

[27] D. Canepa and M. G. Potop-Butucaru. Self-stabilizing tiny interaction protocols. In *WRAS*, pages 10:1–10:6, 2010.

[28] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[29] A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott. Impact of human mobility on the design of opportunistic forwarding algorithms. In *INFOCOM*, pages 1 – 13, 2006.

[30] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[31] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[32] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[33] B. Charron-Bost, M. Hutle, and J. Widder. In search of lost time. *Inf. Process. Lett.*, 110(21):928–933, 2010.

[34] B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[35] J. Cortés, S. Martínez, T. Karatas, and F. Bullo. Coverage control for mobile sensing networks. *IEEE T. Robotics and Automation*, 20(2):243–255, 2004.

[36] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and E. Ruppert. When birds die: Making population protocols fault-tolerant. In *DCOSS*, pages 51–66, 2006.

[37] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The weakest failure detector for message passing set-agreement. In *DISC*, pages 109–120, 2008.

[38] E. W. Dijkstra. Self-stabilization systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1973.

[39] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. of the ACM*, 17(11):643–644, Nov. 1974.

[40] D. Dolev, C. Dwork, and L. J. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.

[41] D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM J. Comput.*, 26(2):418–455, Apr. 1997.

[42] S. Dolev. *Self-stabilization.* MIT Press, 2000.

[43] S. Dolev, A. Hanemann, E. M. Schiller, and S. Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks - (extended abstract). In *SSS*, pages 133–147, 2012.

[44] S. Dolev, R. I. Kat, and E. M. Schiller. When consensus meets self-stabilization. *J. Comput. Syst. Sci.*, 76:884–900, December 2010.

[45] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[46] M. Fischer and H. Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *OPODIS*, pages 395–409, 2006.

[47] M. H. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[48] J. Goguen. Realization is universal. *Mathematical systems theory*, 6(4):359–374, 1972.

[49] R. Guerraoui and E. Ruppert. Even small birds are unique: Population protocols with identifiers. In *Technical Report CSE-2007-04*. York University, 2007.

[50] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[51] S. Hong, I. Rhee, S. J. Kim, K. Lee, and S. Chong. Routing performance analysis of human-driven delay tolerant networks using the truncated levy walk model. In *MobilityModels*, pages 25–32, 2008.

[52] A. Israeli and M. Li. Bounded time-stamps. *Distrib. Comput.*, 6(4):205–209, July 1993.

[53] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In *ASPLOS*, pages 96–107, 2002.

[54] T. Karagiannis, J. L. Boudec, and M. Vojnovic. Power law and exponential decay of inter contact times between mobile devices. In *MOBICOM*, pages 183–194, 2007.

[55] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[56] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[57] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, Dec. 2001.

[58] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[59] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.

[60] L. Lamport. Byzantizing paxos by refinement. In *DISC*, pages 211–224, 2011.

[61] J. R. Lawton and R. W. Beard. Synchronized multiple spacecraft rotations. *Automatica*, 38(8):1359–1364, 2002.

[62] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[63] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1997.

[64] O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Mediated population protocols. *Theor. Comput. Sci.*, 412(22):2434–2450, 2011.

[65] O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Terminating population protocols via some minimal global knowledge assumptions. In *SSS*, pages 77–89, 2012.

[66] R. Mizoguchi, H. Ono, S. Kijima, and M. Yamashita. On space complexity of self-stabilizing leader election in mediated population protocol. *Distributed Computing*, 25(6):451–460, 2012.

[67] A. Mostéfaoui, S. Rajsbaum, M. Raynal, and C. Travers. The combined power of conditions and information on failures to solve asynchronous set agreement. *SIAM J. Comput.*, 38(4):1574–1601, 2008.

[68] A. Mostéfaoui, S. Rajsbaum, M. Raynal, and C. Travers. On the computability power and the robustness of set agreement-oriented failure detector classes. *Distributed Computing*, 21(3):201–222, 2008.

[69] R. Olfati-Saber, J. A. Fax, and R. M. Murray. Reply to "comments on "consensus and cooperation in networked multi-agent systems"". *Proceedings of the IEEE*, 98(7):1354–1355, 2010.

[70] R. Olfati-Saber and J. S. Shamma. Consensus Filters for Sensor Networks and Distributed Sensor Fusion. *44th IEEE Conf. Decision and Control, 2005, and 2005 Eur. Control Conf. (CDC-ECC'05)*, pages 6698–6703, Dec. 2005.

[71] R. Oshman. *Distributed Computation in Wireless and Dynamic Networks*. PhD thesis, MIT, Department of Electrical Engineering and Computer Science, 2012.

[72] W. Ren, R. W. Beard, and E. M. Atkins. A survey of consensus problems in multi-agent coordination. In *American Control Conference*, pages 1859–1864, 2005.

[73] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[74] G. Tel. *Introduction to Distributed Algorithms (2nd ed.)*. Cambridge University Press, 2000.

[75] W. Xi, X. Tan, and J. S. Baras. A stochastic algorithm for self-organization of autonomous swarms. In *Proc. 44th IEEE Conf. Decision and Control, 2005 and 2005 Eur. Control Conf. (CDC-ECC'05)*, pages 765–770, 2005.