Computability Abstractions for Fault-tolerant Asynchronous Distributed Computing

Julien Stainer

N° d'ordre : 00000 ANNÉE 2015

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Julien STAINER

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Systèmes Aléatoires
Université de Rennes 1

Computability Abstractions for Fault-tolerant Asynchronous Distributed Computing

Thèse soutenue à Rennes
le 18 mars 2015

devant le jury composé de :

Petr KUZNETSOV
Professeur à Telecom ParisTech / *Rapporteur*

Sébastien TIXEUIL
Professeur à l'Université de Paris 6 Pierre et Marie Curie / *Rapporteur*

Hugues FAUCONNIER
Maître de conférences à l'Université de Paris 7 Denis Diderot / *Examinateur*

Claude JARD
Professeur à l'Université de Nantes / *Examinateur*

Achour MOSTEFAOUI
Professeur à l'Université de Nantes / *Examinateur*

François TAÏANI
Professeur à l'Université de Rennes 1 / *Examinateur*

Michel RAYNAL
Professeur à l'Université de Rennes 1, Membre Senior de l'Institut Universitaire de France / *Directeur de thèse*

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie LAMPORT

Remerciements

Je remercie Petr KUZNETSOV, Professeur à Telecom ParisTech, et Sébastien TIXEUIL, Professeur à l'Université de Paris 6 Pierre et Marie Curie, d'avoir bien voulu accepter la charge de rapporteur.

Je remercie Hugues FAUCONNIER, Maître de conférences à l'Université de Paris 7 Denis Diderot, Claude JARD, Professeur à l'Université de Nantes, Achour MOSTEFAOUI, Professeur à l'Université de Nantes, et François TAÏANI, Professeur à l'Université de Rennes 1, d'avoir bien voulu juger ce travail.

Je remercie Michel RAYNAL, Professeur à l'Université de Rennes 1, Membre Senior de l'Institut Universitaire de France, qui a dirigé ma thèse. Je considère comme un honneur d'avoir pu collaborer avec lui. Son travail, son humour et sa vision de notre domaine constituent pour moi un exemple et une source d'inspiration essentielle.

Je remercie tous mes co-auteurs et plus généralement les membres la communauté de l'informatique distribuée pour tous les échanges enrichissant que j'ai pu avoir avec eux durant cette thèse.

Je remercie tous mes collègues de l'équipe ASAP pour l'excellente ambiance de travail et le cadre stimulant dans lesquels j'ai eu le plaisir de travailler durant cette thèse.

Je remercie Matoula, tous mes amis et ma famille pour m'avoir supporté (dans tous les sens du terme) jusqu'ici.

Je remercie enfin les membres de la communauté du logiciel libre pour l'esprit de partage et d'entraide dont ils font preuve et pour la qualité des logiciels qu'ils mettent à disposition.

Résumé en Français

Introduction et Motivations

Modèles de base Cette thèse porte sur la calculabilité dans les système distribués asynchrones sujets aux défaillances. Plusieurs modèles représentant ce type de systèmes sont étudiés au long de ce document. Les modèles de base sont constitués de processus séquentiels asynchrones communicant par passage de messages ou par mémoire partagée. Dans ces modèles, il n'y a pas de bornes sur les vitesses relatives entre processus. Sauf indication particulière, c'est la version $wait\text{-}free$ de ces systèmes qui est prise en considération ; c'est-à-dire que parmi n processus, jusqu'à $n-1$ peuvent subir une défaillance.

Détecteurs de fautes Dans les modèles asynchrones et wait-free, certaines tâches sont impossible à résoudre, essentiellement à cause de l'impossibilité pour les processus de distinguer un processus extrêmement lent d'un processus ayant subi une défaillance. Ainsi, un des problèmes fondamentaux du calcul distribué, le $consensus$ est impossible à résoudre dans ces systèmes [31, 57]. Le consensus consiste, pour les processus, à se mettre d'accord sur une valeur proposée par l'un d'entre eux. Afin de définir, de façon modulaire, de nouveaux modèles, basés sur les modèles asynchrones classiques, dans lesquels ces problèmes ont des solutions, la notion de détecteur de fautes [20, 56] à été introduite. Il s'agit de supposer qu'un dispositif, contrôlé par le système, fournit aux processus de l'information sur les défaillances survenant dans l'exécution considérée.

L'exemple le plus connu de détecteur de fautes est Ω, qui fournit à tout moment à chaque processus l'identité d'un $guide$ (l'un d'entre eux). Les guides des processus peuvent prendre des valeurs arbitraires pendant un temps non borné, mais après un temps fini, tous les processus ont le même guide jusqu'à la fin de l'exécution et il s'agit d'un processus n'ayant pas subi de défaillance. Il a été prouvé dans [19] qu'Ω est à la fois nécessaire et suffisant pour résoudre le consensus dans les modèles wait-free asynchrones dans lequels les processus communiquent par mémoire partagée.

Modèles itérés Une des principales difficultés levées par l'étude de l'ensemble des exécutions possibles dans les modèles asynchrones réside dans le nombre extrêmement élevé d'entrelacements possibles entre les étapes prises par les différents processus. Afin de réduire cette complexité, plusieurs modèles itérés tels que le modèle du $snapshot\ immediat\ itéré$ [14] ou le modèle synchrone sans fautes à passage de messages affaibli par un adversaire [2] ont été proposés. Dans ces modèles, les processus exécutent une séquence de rondes et communiquent au court d'une ronde uniquement avec les processus exécutant cette même ronde. Ceci mène à un schéma de communication beaucoup plus contraint dont la structure est plus simple et plus régulière.

Il est prouvé dans [14] que le modèle du snapshot immediat itéré permet de résoudre les mêmes tâches que le modèle wait-free asynchrone dans lequel les processus partagent de la mémoire. Parallèlement, il est montré qu'en affaiblissant le modèle synchrone sans fautes à passage de messages avec un des adversaires proposés par [2], on obtient également un modèle équivalent au modèle

de la mémoire partagée asynchrone wait-free. Dans les deux cas, la structure des exécutions possibles dans ces modèles itérés est plus régulière et a donné lieu à une étude à travers le prisme de la topologie combinatoire [45, 2].

Généralisations du problème du consensus Le problème du consensus a été généralisé de deux façons afin de relâcher la contrainte de l'unicité de la valeur décidée. L'une de ces façons a donné naissance au problème du $k\text{-}accord\ ensembliste$ [22] dans lequel les processus proposent chacun une valeur et décident chacun une des valeurs proposées, sous la contraite qu'au plus k valeurs distinctes soient décidées dans tout le système. La seconde façon a été décrite par [4] et est désignée par le nom de consensus s-simultané. Pour résoudre ce problème, chaque processus propose un vecteur de s valeurs et doit décider une paire (idx, v) où idx est un index compris entre 1 et s et v est l'une des valeurs proposées en idx^e position de leurs vecteurs par les processus. Si deux processus décident des paires (idx, v) et (idx', v'), alors $idx = idx' \Rightarrow v = v'$.

Dans les systèmes wait-free asynchrones dans lesquels les processus partagent de la mémoire, ces deux généralisations constituent des problèmes équivalents [4]. Cependant, [17] montre que ce n'est plus le cas lorsque l'on considère les systèmes où les processus communiquent par passage de messages.

Adapter les détecteurs de fautes aux modèles itérés

La première contribution de ce document présente des cadres génériques pour construire des modèles itérés dérivés de ceux décrits dans l'introduction de façon à obtenir des modèles pouvant calculer les mêmes tâches que les modèles de base enrichis par certains détecteurs de fautes connus. Les deux constructions s'appuient sur une notion commune de processus $fortement\ corrects$ décrivant l'ensemble des processus qui, au cours d'une exécution donnée, ont infiniment souvent la possibilité de faire parvenir un message, directement ou non, à tous les autres processus.

Pour le modèle du snapshot immediat itéré, une façon syntaxique de transformer les détecteurs de fautes usuels en détecteurs de fautes utilisables dans ce modèle itéré est présentée. Des algorithmes et preuves génériques pour prouver l'équivalence du système obtenu et du système de base enrichi par le détecteur de fautes d'origine sont introduits. Ce travail a été publié dans [81].

Dans le cas du modèle synchrone sans fautes à passage de messages, de nouveaux adversaires correspondant à certains détecteurs de fautes sont proposés. Lorsque le modèle est affaibli par l'un d'eux, les tâches que l'on peut y résoudre sont les mêmes que celles que l'on peut résoudre dans l'un des systèmes de base enrichi par le détecteur de fautes correspondant. Cette partie a donné lieu à une publication [82].

Du k-accord ensembliste au consensus s-simultané

Il a été montré dans [17] que les problèmes du k-accord ensembliste et du consensus s-simultané, bien qu'équivalents dans les systèmes wait-free asynchrones où les processus partagent de la mémoire, ne le sont plus lorsque la communication se fait par passage de messages. La seconde contribution de ce document montre, que l'on peut même définir une famille de problèmes qui généralise les deux précédents et dont la difficulté varie.

Des détecteurs de fautes suffisant, et d'autre nécessaires, à résoudre ces problèmes sont présentés. Ils servent d'outils à l'étude des relations entre les différents problèmes de la hiérarchie et leur niveau de difficulté. Les travaux présentés dans cette partie ont été publiés dans [84].

Construction universelle basée sur des consensus simultanés

Le problème du consensus est essentiel dans les systèmes distribués, car lorsque l'on sait le résoudre, on peut l'utiliser comme brique de base pour implémenter de façon résiliente n'importe quel objet partagé ayant une spécification séquentielle. Ce type de construction est appelé *construction universelle*. Une façon d'étendre ce résultat dans dans systèmes où le consensus n'est pas possible a résoudre mais où le consensus s-simultané est possible a récemment été proposé par [34]. Elle autorise l'implémentation de s objets dont au moins un progresse.

Cette troisième contribution revisite leur construction, y propose une alternative plus modulaire offrant la possibilité d'offrir des conditions de progression plus fortes pour les opérations effectuées sur les objets partagés. Certains défauts de la proposition d'origine sont corrigés et les propriétés de la nouvelle construction sont étudiées. Elle propose par exemple une façon d'éviter l'utilisation des objets consensus s-simultané et de garantir le progrès de tous les objets lorsque l'on se trouve en l'absence de contention.

Calculabitité en présence d'executions isolées concurrentes

Finalement, la quatrième contribution de ce document présente une nouvelle famille de modèles itérés autorisant un nombre maximal fixé d de processus s'exécutant en isolation des autres de façon concurrente. Lorsque $d = 1$, le modèle obtenu correspond exactement au modèle du snapshot immediat itéré et le système obtenu peut donc calculer les mêmes tâches que le système wait-free asynchrone où les processus communiquent par mémoire partagée. À l'autre extrême, lorsque $d = n$, tous les processus peuvent avoir à décider en se basant uniquement sur leur propre valeur d'entrée, les tâches qui peuvent être calculées dans ce modèle correspondent donc à celle calculables dans un système wait-free asynchrone dans lequel les processus communiquent par passage de messages.

Une nouvelle famille de problèmes est également introduite dans cette partie et est utilisée pour prouver que pour tout $d < n$, le modèle paramétrisé par d peut résoudre strictement plus de tâches que celui associé au paramètre $d + 1$. Cette nouvelle hiérarchie de modèles itérés est donc stricte en termes de calculabilité. Enfin, une étude de la structure de l'ensemble des exécutions possibles dans ces modèles est menée à l'aide de notions de topologie combinatoire. Le travail présenté dans cette partie a été publié dans [43].

Conclusion

Sujets abordés durant cette thèse

Dans ce document plusieurs abstractions, problèmes, algorithmes et simulations sont introduits dans l'espoir de mieux comprendre, en termes de calculabilité, les relations entre les détecteurs de fautes, les systèmes sujets au partitionnement et les modèles itérés.

k-accord ensembliste et consensus s-simultané Les problèmes du k-accord ensembliste et du consensus s-simultané jouent un rôle central dans ces travaux. Une construction universelle basée sur eux est présentée et leurs différences dans le cas où les processus communiquent par passage de messages sont étudiées et donnent lieu à la définition d'une hiérarchie de problèmes intermédiaires.

Détecteurs de fautes Les détecteurs de fautes servent d'outils dans l'étude de cette hiérarchie de problèmes. Par ailleurs, des modèles itérés équivalents aux modèles de base enrichis par des détecteurs de fautes connus sont définis. Cependant, la question du plus faible détecteur de faute pour

résoudre le k-accord ensembliste dans les systèmes wait-free asynchrones à passage de messages reste une question ouverte.

Systèmes sujets au partitionnement Les systèmes sujets au partitionnement sont étudiés sous plusieurs angles au cours de ce document. En considérant que le consensus s-simultané peut être résolu dans certains de ces systèmes, une construction pour y implémenter des objets partagés est proposée. Une nouvelle famille de modèles itérés capable de considérer plusieurs processus s'exécutant seuls en isolation est introduite, ce qui peut être considéré comme une première étape vers un modèle itéré prenant en compte la possibilité de partitionnement.

Modèles itérés Les études qui concernent les modèles itérés dans ce document ont mené à la définition de la notion de processus fortement corrects. Ces processus qui peuvent communiquer infiniment souvent, directement ou non, avec tous les autres jouent un rôle important dans les simulations entre les modèles itérés pris en considération et les modèles non-itérés asynchrones enrichis par des détecteurs de fautes. La nouvelle famille de modèles itérés introduite dans ce document autorise un nombre borné a priori de processus à s'exécuter en isolation de façon concurrente. Les tâches qui peuvent être résolues dans ces modèles varient, en fonction de cette borne, de celles qui peuvent être résolues dans les systèmes wait-free asynchrones équipés de mémoire partagée à celles que l'on peut résoudre dans les systèmes wait-free asynchrones où les processus communiquent par passage de messages. La structure des exécutions possibles dans ces modèles admet une représentation topologique simple.

Autres publications au cours de cette thèse

En plus des travaux présentés dans ce document [81, 85, 84, 87, 43], cette thèse a donné lieu à d'autres publications couvrant plusieurs autres sujets.

La quête du plus faible détecteur de faute permettant la résolution du k-accord ensembliste dans les systèmes wait-free asynchrones dans lesquels les processus communiquent par passage de messages a conduit à une étude des relations liant les détecteurs de fautes ayant été proposés pour ce problème [66]. Elle a aussi généré la proposition d'un nouveau détecteur de fautes [67] permettant la résolution de ce problème tout en étant plus faible que ceux étudiés jusque là. Le détecteur utilisé dans l'étude des relations entre le k-accord ensembliste et le consensus s-simultané est basé sur ce travail.

Deux algorithmes basés sur Ω et résolvant le consensus ont été proposés, l'un utilisant des objets nommés ensembles fermant [82] et l'autre se basant sur des objets store-collect [83]. Une solution pour le problème de la diffusion dans les systèmes dynamiques récurrents dont les canaux de communication ne sont pas fiables a été publiée dans [86]. Dans le contexte des systèmes asynchrones à passage de messages sujets à des fautes byzantines, une proposition de construction de registres atomiques fiables est parue dans [47]. La façon dont les groupes de processus peuvent être utilisés pour réduire l'espace de noms dans le problème du renommage a été explorée dans le cadre des systèmes asynchrones dans lesquels les processus partagent de la mémoire [18]. Enfin j'ai collaboré à la version journal d'un article proposant d'exploiter la notion de confiance exprimée par les réseaux sociaux explicites afin de construire un protocole d'échantillonnage aléatoire des pairs dans les systèmes pair-à-pair [32].

Contents

Chapter 1

Introduction and Models

1.1 Introduction

In the last decades, computer systems evolved from centralized single processor units running local programs to interconnected swarms of multi-core machines running communicating processes. Even in the case of single machine local computations, in order to benefit of the multi-core architecture of modern processors, the applications have to be split up into several threads or processes. Communication between these entities is needed in order to achieve the application goal collaboratively and to access shared data in a coherent manner. This already rises numerous issues related to asynchrony, scheduling, mutual exclusion and progress guarantees. To be able to produce correct and efficient single machine programs and systems, these problems have to be tackled by leveraging the hardware primitives multi-core architectures offer, to synchronize the multiple agents involved in the computation.

Nowadays, however, in addition of the fundamental problematics of single machine synchronization, additional issues became essential. Some web applications run on heterogeneous systems composed of hundreds of multi-core machines. These machines can be distributed across several datacenters, some of them can be virtualized and the network connecting them can be prone to message delays or losses. In these systems, the level of asynchrony and the frequency of failures are high and the only way to achieve the desired performance and availability is to carefully design algorithms and protocols able to handle the unreliability and the heterogeneity of the underlying platforms.

This leads us to an era of computing in which concurrency, asynchrony and failures are essential parameters to be able to develop large scale applications on top of complex heterogeneous unreliable platforms. Research on distributed systems provided us with fundamental understanding of theoretical impossibilities and algorithms that helped both designing and programming large scale heterogeneous crash-prone systems.

1.2 Models and Fundamental Results

In this thesis, a *distributed system* designates a set of computing entities called *processes* that communicate. In terms of sequential computability, the processes are considered to be Turing machines, provided with additional communication operations. From a practical point of view, the notion of process can modelize several real-world entities such as the cores of a multi-core that communicate through the bus of a processor, the threads scheduled by the operating system to run on these cores that share memory offered by the system, or even distant computers communicating through messages sent on a network.

The algorithms for these systems describe a set of programs designed to run on each process. The global input of these algorithms are not, contrarily to the sequential situation, directly available to each process. In the case of distributed computations, each of them is given access to its own input and it discovers the other inputs only by communicating with the other processes.

In sequential computing, the notion of function captures the abstraction of computation. The specification of a function describes its possible inputs, outputs and the relation giving the set of allowed outputs for each possible input. This specification is independent of both the model of computation used and of the practical sequence of operations used to solve it, its implementation.

In distributed systems, the analogous of a function is a *task*. Similarly as in the sequential case, it is specified by providing the set of possible *input configurations*, *output configurations* and the relation associating the set of allowed output configurations to each input configuration.

Providing an algorithm that solves a task in a distributed system model means giving to each process of a system a program whose input is the local input of the process and that, with any global input configuration given by the task specification and in any possible execution allowed by the model, produces an output such that, the global output configuration of the system is one of those allowed by the relation of the task specification when considering this particular input.

Among distributed system models, two families can be distinguished: the *synchronous* ones in which all the processes in the system progress in lock-step and *asynchronous* ones in which each of them progresses at its own pace, unknown to the rest of the system. This thesis mainly consider asynchronous systems that, however harder to design for, allow to leverage heterogeneity and naturally handle crash failures as extreme slowness.

In asynchronous systems, an execution may be represented as a sequence of events modelizing the order in which processes take steps, these steps being local computing steps or interactions with the communication medium. These sequences of steps are called *schedules*.

The models considered in this thesis take into account the occurrence of failures. Several types of failures are considered, from the *crash* of some processes during which they just stop taking steps, to the *Byzantine* failures representing that some processes can behave arbitrarily. During an execution, a process that crashes or exhibits a Byzantine behavior is said *faulty*, while the other processes are called *correct*.

The extreme case of models in which up to $n - 1$ processes may crash during some computations is called (ambiguously) *wait-free*. In an asynchronous wait-free model, if no additional information is provided to the processes, it is impossible for a process to wait for others. The *wait-free* qualifier is also sometimes used to describe a completely orthogonal notion which is a *progress condition*.

In sequential computing, the notion of object allows to encapsulate computing abstractions as self-contained entities storing their own data and offering operations to access and modify their state. The same definition can be used in the context of distributed computing. A *distributed object* thus describes an abstract entity whose allowed behavior is described by a formal *specification*. Such an object offers to the processes in the system some primitives that allow them to query and modify the logical state of that object. The underlying implementation of these primitives may involve using the basic communication primitives of the system.

For distributed objects whose specification is sequential, this thesis considers a useful property of implementations which is *linearizability* [46]. An implementation of an object is linearizable if there is a sequential history of the operations invoked by the processes such that each of these operations appear as executed instantaneously at a given point in time between its invocation and the moment it returns. Linearizability is *composable*, meaning that, when used by sequential processes, a set of objects implemented in a linearizable manner is linearizable [46].

1.2.1 Message-passing systems

One of the classic models for communication in distributed systems is the message-passing model. Processes are connected with each other by *channels* composing a network. They are provided with a communication primitive send allowing them to send a message over one of the (potentially unidirectional) channels that originate from them. On the other side of that channel, after the message arrives, an interruption is raised on the connected process, to which the content of the message is then delivered and which can react accordingly.

According to the considered model, the channels can be *reliable* or not, according to the fact that they may lose messages that they were supposed to carry. The channels can also be *synchronous*, meaning that there is a bound on the time between the sending and the reception of a given message, or *asynchronous* if that delivery time is unbounded (in that case, if the channel is reliable, these delivery delays are all finite, despite the fact that no upper bound on them is known). In this thesis, we consider that a message that is delivered is not modified in transit. In practice, this can be ensured by the use of checksum mechanism for example.

1.2.2 Shared memory systems

Another classic inter-process communication mechanism is the use of a shared memory composed of a set of shared *registers*. A register is a shared object storing some information and offering to the processes two elementary primitives, read and write, respectively allowing them to access the content of the register and to modify it. Several types of registers have been studied in the literature [52], but even in the case of asynchronous crash-prone systems, it has been shown that *atomic* (also said linearizable) registers can be implemented from *safe* registers that are the weakest classic registers usually considered [51]. Similarly, the amount of information that can be stored in a register is not really relevant in terms of computability. Indeed, even in the case of asynchronous crash-prone systems, unbounded registers (that can store an arbitrarily large amount of information) can be implemented from binary ones. Consequently, this thesis only considers atomic unbounded registers.

Another property that characterizes a register is the sets of processes that can write and read to and from it. Several variations of that property have been studied, from single-writer single-reader (SWSR) registers that are used unidirectionally for the communication from one process to another, to multi-writers multi-readers (MRMW) registers sometimes allowing that any process can read or write them. This thesis focuses on single-writer multi-readers (SWMR) registers that only one process can write but that can be read by any of them.

Several useful abstractions have been built on top of these of atomic single-writer multi-reader registers. The *snapshot* primitive allows a process to take an image of the content of the whole set of registers in an atomic fashion [1]. This primitive can be wait-free implemented in shared memory systems in the presence of an arbitrary number of crash failures.

Another useful abstraction that can be built in these systems: the *immediate snapshot* object [13]. It is specified as providing a single *write-snapshot* operation that allows a process to write a value and retrieve a snapshot of the entire memory. The *immediate snapshot* object is said *one shot* because each process is supposed to call *write-snapshot* at most once. The concurrent invocations of this operation are *set linearizable* [69]. Two of them can either appear to (i) happen one after the other, in which case the first snapshot does not contain the value written during the second operation, and the second snapshot contains the first one or (ii) happen simultaneously, both snapshots being then identical and containing both written values. The implementation respects the real-time order: if an invocation of *write-snapshot* ends strictly before the beginning of another, then case (a) with the operations in the same order is the only allowed behavior.

The *immediate snapshot* object has been instrumental in studying distributed computability

in asynchronous crash-prone shared memory systems. The *iterated immediate snapshot* model considers asynchronous processes that communicate only through a sequence of *immediate snapshot* objects. The execution is stripped into asynchronous rounds during which each process uses the *write-snapshot* operation of the object associated to the round to write its state and retrieve a snapshot of the other processes states that have already been written in the object at that point. It then computes its next state and proceeds to the next round. The obtained model strongly constrains the communications between processes but can be simulated in the classic asynchronous shared memory model prone to an arbitrary number of failures. In that model, the structure of the possible states of the processes in all the allowed executions is regular and can be studied through combinatorial topology [45].

Moreover, the *iterated immediate snapshot* has been shown to have the same computational power, with respect to task solvability, than the classic asynchronous wait-free shared memory model [14]. This equivalence brought us a celebrated topological characterization of asynchronous computability [42, 45].

1.2.3 Failure detectors

In order to design failure tolerant algorithms, it is sometimes needed to assume that some information on failures is available to the processes. This additional assumption is often abstracted by the notion of *failure detector* introduced by [20, 56]. Enriching a model with a failure detector can be thought as providing each process with a black-box type device that controls a variable accessible by the process in a read-only fashion. The specification of the failure detector defines the values that can be output by these devices according to the pattern of failures in a given execution.

A major advantage of this approach is its modularity. It gives the ability to design algorithms that suppose that a given information on failures is available to the processes, independently from the way it is gathered. Additionally, it allows to propose algorithms that rely on an eventually accurate information on failures to terminate, but that tolerate an arbitrarily long period of somehow inaccurate information without risking to violate their specification.

An example of failure detector is Ω, the *eventual leader* failure detector. It provides each process with a variable, which it can consult at will, containing a process identity. These variables can contain arbitrary identities for an unbounded period of time, but eventually, the variables of all the correct processes stop evolving and contain the same correct process identity.

In a given model M, it is possible to define a partial order on failure detectors. A failure detector A is said *weaker* than another failure detector B in the considered model M if and only if there exists an algorithm that implements A in M enriched with the information on failures provided by B. If A is weaker than B and B is weaker than A then the two of them are said *equivalent* in the model M.

When considering a problem P in a given model M, a failure detector A is said to be the *weakest failure detector* associated to P if and only if (i) there is an algorithm solving P in M enriched by the information on failures provided by A and (ii) A is weaker than any failure detector B such that there exists an algorithm solving P in M enriched by B. It is proved in [48] that any problem solvable with a failure detector has an associated weakest failure detector.

1.3 Notations

This section introduces the notations used in this thesis. The notions they describe have been introduced in section 1.2.

The considered systems are made of n processes denoted $p_1, \ldots, p_n$. The set $\{p_1, \ldots, p_n\}$ is referred to as Π. During a given execution, the set of processes that fail is denoted $\mathcal{F}$ while

$\Pi \setminus \mathcal{F}$, the set of correct processes, is denoted $\mathcal{C}$. When a computation in a model is simulated in another model, the simulator processes of the underlying model are also denoted $q_1, \ldots, q_n$, while the processes of the simulated model are designated by $p_1, \ldots, p_m$.

The asynchronous message-passing and asynchronous shared memory models are respectively denoted $\mathcal{AMP}$ and $\mathcal{ARW}$, while $\mathcal{SMP}$ designates the synchronous message-passing model. Additional precisions on the considered models may be added in two ways: a subscript giving the size of the system and the maximal number of failures as in $\mathcal{ARW}_{n,t}$ or other information on the model given between square brackets, such as a failure detector provided to the processes or a message adversary i.e. $\mathcal{ARW}[\Omega]$ to describe the asynchronous shared memory model enriched with the failure detector Ω.

1.4 Impossibilities and Known Workarounds

This section considers two fundamental impossibilities in distributed systems. The first one shows that, in asynchronous systems where half of the processes or more can crash in some executions, communication via message-passing does not allow to simulate a shared memory. The second one shows that in asynchronous systems prone to one crash or more, the consensus problem is impossible to solve.

Insights on the reasons of these impossibilities, as well as known workarounds are presented in the following.

1.4.1 From Message-Passing to Shared Memory

Implementing a shared memory in asynchronous message-passing systems rises some issues. To be able to enforce the linearizable semantics of atomic registers, the processes have to make sure of two things: they have to ensure that after the end of a write operation on a register, no process will ever read an old value from that register and after the end of a read operation on a register returning a given value, no process will read an older value from that register.

In systems of n processes where strictly less than $\frac{n}{2}$ crashes can happen in a given execution, this can be enforced by two mechanisms. When writing, a process waits that at least $\lceil \frac{n+1}{2} \rceil$ processes acknowledge the fact that they took the write operation into account. When reading, a process waits for at least $\lceil \frac{n+1}{2} \rceil$ processes to communicate the freshest values that have been written to the register from their point of view, and then it writes it, following the previously described procedure.

A natural question follows: how to implement a shared memory in a message-passing system in which more than half of the processes can fail? [24] proposes an approach based on failure detectors and shows that Σ is the weakest failure detector that allows to implement a shared memory in such a system. Informally, Σ provides to each process a, possibly always changing, read-only variable containing a set of process identities called $quorum$. These sets verify two properties: any two of them, taken at any moment of the execution and on any process, always intersect and eventually they only contain correct processes.

1.4.2 Solving Consensus

A fundamental problem in distributed systems is the need for the different processes composing a system to $agree$. These processes may have different local views of the global state of the system but numerous algorithms require that they find an agreement, for example on the next operation to apply on a simulated shared object, on a channel on which further communication should occur, etc.

This agreement task has been studied in depth in the literature [31, 37, 38, 53, 54, 71]. It is formalized as a task under the denomination of $consensus$. Namely, each process participating to the consensus proposes a value and, if it does not fail, has to output (also said $decide$) a value. All the processes that decide have to output the same value, and it has to be one of the proposed values.

Several algorithms solving consensus in different models have been proposed. However, in their celebrated paper [31], Fisher, Lynch and Patterson show that, in asynchronous systems, as soon as one failure can occur, solving consensus is impossible. Interestingly, this result is independent of the way processes communicate with each other and thus holds in shared memory systems [57] as well as in message-passing ones [31].

In order to overcome that impossibility, several approaches have been proposed. Some consider reducing the space of the values that can be proposed [63], some rely on stronger communication objects being available [38], some other are based on failure detectors [56, 82, 83]. It is shown in [19] that, in the asynchronous shared memory models, the failure detector Ω is necessary and sufficient to solve consensus, while [25] proves that the combined failure detectors Σ and Ω are necessary and sufficient in asynchronous message-passing systems.

Intuitively, a shared memory or the failure detector Σ is needed to prevent the system from partitioning in two independent ones that could evolve separately and thus could decide different values, which violates the consensus specification. Besides this safety requirement, the failure detector Ω plays a role in the liveness of the consensus algorithms.

1.4.3 Generalized Consensus: the k-Set Agreement Problem

The k-set agreement problems The k-set agreement problem is a paradigm of coordination problems. Defined in the setting of systems made up of processes prone to crash failures, it is a simple generalization of the consensus problem (that corresponds to the case $k = 1$). The aim of this problem, introduced by Chaudhuri [22], was to investigate how the number of choices (k) allowed to the processes is related to the maximum number of processes t that can crash. The problem is defined as follows. Each process proposes an input value, and any process that does not crash must decide a value (termination), such that a decided value is a proposed value (validity), and no more than k distinct values are decided (agreement).

While it can be solved in synchronous systems prone to any number of crashes (see [77] for a survey), the main result associated with k-set agreement is the impossibility to solve it in presence of both asynchrony and process crashes when $t \geq k$ [13, 45, 89].

A way to circumvent this impossibility consists in enriching the underlying pure asynchronous system with a failure detector [20, 88]. A failure detector is a device that provides processes with information on failures. According to the type and the quality of this information, several failure detectors have been proposed (see [78] for a survey of failure detectors suited to k-set agreement). It has been shown that the failure detector $\overline{\Omega}_k$ (anti-omega-k) [75, 96] is the weakest failure detector that allows k-set agreement to be solved despite any number of process crashes in asynchronous *read/write* systems [35].

The situation is different in asynchronous crash-prone *message-passing* systems. More precisely, (a) while weakest failure detectors are known only for the cases $k = 1$ and $k = n - 1$ [19, 24, 26], (b) it has been shown that the generalized quorum failure detector denoted Σ_k is necessary [12]. k-Set agreement algorithms based on failure detectors stronger than Σ_k can be found in [12, 16, 65, 66, 67].

The s-simultaneous consensus problem This problem has been introduced in [4]. Each of the n processes proposes the same value to s independent instances of the consensus problem, denoted

$1, ..., s$. Each correct process has to decide a pair (c, v) (termination), where $c \in \{1, ..., s\}$ is a consensus instance and v is a proposed value (validity). Moreover, if (c, v) and (c, v') are decided we have $v = v'$ (agreement). (This is the scalar form of the problem: each process proposes the same value to each consensus instance. In the vector form, a process proposes a vector of s values, one value to each consensus instance. It is shown in [4] that both forms have the same computational power.)

It is shown in [4] that the x-simultaneous consensus problem and the x-set agreement problem are computationally equivalent in asynchronous read/write systems where up to $t = n - 1$ processes may crash. It follows that in these systems, the failure detector $\overline{\Omega}_x$ is both necessary and sufficient to solve x-simultaneous consensus.

As far as asynchronous message-passing systems are concerned, it is shown in [17] that, for $x > 1$ and $t > \frac{n+x-2}{2}$, x-simultaneous consensus is strictly stronger than x-set agreement. This means that, differently from what can be done in asynchronous read/write systems, it is not possible to solve x-simultaneous consensus from a black box solving x-set agreement.

1.5 Motivations and Contents

This PhD initially focused on a computability problem that has been studied for several years, namely, finding the weakest failure detector for an asynchronous wait-free (as a resiliency condition) message-passing system that allows to solve the k-set agreement in a wait-free (as a progress condition) manner. Even though we did not succeed at finding that weakest failure detector, we managed to compare and relate the different failure detectors that had been previously proposed [66] and we defined a new one that still allows k-set agreement to be solved while being generally strictly weaker that the existing proposals [67].

Facing difficulties to improve this result toward optimality, we felt the need to better understand both failure detectors and the k-set agreement problem. To achieve this goal, we tried to study how the notion of failure detectors could be translated in the context of two iterated models: the iterated immediate snapshot model and the message-adversary model. This work is presented in Chapter 2. We also studied how the k-set agreement problem compares to the closely related problem of k-simultaneous consensus and some of its variants. We show in Chapter 3 that these problems constitute a hierarchy when considering asynchronous wait-free message-passing systems while they are all equivalent in the case of wait-free shared memory systems.

In Chapter 4, we present a universal construction based on k-simultaneous consensus objects and atomic shared registers that extends [34]. We propose a modular algorithm that builds several concurrent objects that can be accessed by the processes, guaranteeing that at least one of them will provide wait-free operations forever. We also investigate the possibility of ensuring the progress of more of the objects by enriching the base shared memory system with a stronger version of k-simultaneous consensus.

As shown in Chapter 2, while equivalent in asynchronous shared memory wait-free systems, the k-set agreement and k-simultaneous consensus problems split into a strict hierarchy of agreement problems when message-passing communication replaces the shared memory. In order to better capture the differences between these two systems that differ on the communication medium used, we proposed a new computation model that generalizes both shared memory and message-passing communications in the context of wait-free asynchronous models. In Chapter 5 we present this new abstraction named d-solo model and study the tasks that can be solved in it. We show that by varying the number of processes that can run alone without hearing from any other process, we obtain a strict hierarchy of models that spans from the shared memory model to the message-passing one.

In Chapter 6, we discuss how the presented studies can be extended to further the understanding of k-set agreement in message-passing systems and more generally of distributed objects implementations in potentially partitioning crash-prone systems. We then open the perspective on other works done during this PhD and conclude.

Chapter 2

Failure Detectors in Iterated Models

This chapter studies how the notion of failure detector can be ported to two iterated models: the iterated immediate snapshot model and the synchronous message-passing model weakened by message-adversaries. The results presented here have been published in [81, 82]. Section 2.1 gives the formal specification of the iterated immediate snapshot model, denoted $\mathcal{IIS}$, Section 2.2 presents a systematic way to transform failure detectors suited for the asynchronous shared-memory model into ones that fit the $\mathcal{IIS}$ model and Sections 2.3-2.4 give a framework to prove that the transformation preserves computability. Finally Section 2.5 studies the cases in which the maximal number of failure is restricted.

Section 2.6 presents the notion, introduced in [2], of message-adversaries in failure-free synchronous message-passing models and the main result associated to that notion. Sections 2.7-2.10 introduce message-adversaries such that, when weakened by these, the synchronous message-passing model has the same computability power, with respect to colorless tasks, than the wait-free asynchronous message-passing or shared memory models enriched with some well studied failure detectors. Finally, Section 2.11 concludes the chapter.

2.1 The Iterated Immediate Snapshot Model

As described in 1.2.2, in the $\mathcal{IIS}$ model, the processes execute asynchronous rounds and, during each of these rounds, communicate through a one-shot immediate snapshot object associated to the round. As stated before, the immediate snapshot object provides the processes with a unique operation: write_snapshot(v), where v is the value the process wants to write in the memory of the object. Let us remember that each process can invoke that operation only once. When it does so, the value v is written in the object and a snapshot containing the values already written is immediately taken and returned to the process.

The invocations of the operation write_snapshot(v) on a given immediate snapshot object by the different processes are set-linearizable [69]. If we denote by v_i the value written by a process p_i and by $view_i$ the view that it retrieves from the immediate snapshot object, then the following properties are verified:

- Self-inclusion. $\forall\, i :\ \langle i, v_i\rangle \in view_i$.
- Containment. $\forall\, i, j :\ (view_i \subseteq view_j)\ \vee\ (view_j \subseteq view_i)$.
- Immediacy. $\forall\, i, j :\ (\langle i, v_i\rangle \in view_j) \Rightarrow\ (view_i \subseteq view_j)$.

The first property states that a process gets a view that contains its own value, the second one shows that the views retrieved by different processes are ordered (they cannot be incomparable),

while the last one ensures that a process p_j that retrieves a view containing the value written by p_i gets a view that contains the one that p_i got. A consequence of these rules is that two processes that see each other get exactly the same view; in other words, their operations are set-linearized together, they appear as executed at the same moment.

As stated in Introduction 1.2.2, it is possible to implement a one-shot immediate snapshot object in a wait-free manner in the asynchronous shared memory model prone to any number of failure without any additional assumption [14]. Consequently, it is also possible to simulate the iterated immediate snapshot model in that base model. Reciprocally, [14] also shows that it is also possible to simulate a wait-free shared memory system in the iterated immediate snapshot model. Both models are consequently computationally equivalent.

2.2 Porting the notion of failure detector to the $\mathcal{IIS}$ model

On one hand, the notion of failure detector has been instrumental in describing precisely how to reinforce the wait-free shared memory model in order to be able to solve fundamental problems such as consensus. They offer an accurate way of stating the hypothesis needed to solve these problems and are thus a nice tool to study computability. On the other hand, the executions in the iterated immediate snapshot model are more constrained than those in the base asynchronous shared memory model, even if the computability power of these models is the same. $\mathcal{IIS}$ consequently offers a framework in which it is easier to reason. In addition, the regular structure of the executions of a protocol in $\mathcal{IIS}$ allows us to represent them using combinatorial topology [42, 45], enabling the use of powerful mathematical tools to study computability.

The aim of this chapter is to get the best of both worlds, namely to be able to keep the simple and powerful way of refining models through the use of failure detectors, while continuing to restrict ourselves to the regularity and structure of $\mathcal{IIS}$. However, it has been shown in [73] that if a task is solvable with a failure detector in $\mathcal{IIS}$, then it is also wait-free solvable in $\mathcal{IIS}$ without any failure detector. This result thus essentially states that failure detectors do not bring any additional computability power to $\mathcal{IIS}$. The same authors proposed in [74] a constrained version of the $\mathcal{IIS}$ model whose computability power matches the $\mathcal{IIS}$ model enriched with the failure detector $\diamond S_x$; however, their construction relies on designing an additional synchrony property to restrict the possible executions in $\mathcal{IIS}$ and then on simulations designed for those specific failure detector and property in order to show the computability equivalence. In contrast, the approach presented here aims at having a generic way to modify a failure detector used in $\mathcal{IIS}$ in order to obtain a failure detector suited for $\mathcal{IIS}$ and proposes a general framework to build and prove the matching simulations showing that the computability properties are preserved.

2.2.1 Strongly Correct Processes in $\mathcal{IIS}$

This section introduces a notion that is instrumental in the entire chapter, the notion of *strongly correct processes*. Intuitively, the need for a stronger notion of what a correct process is comes from the following observation: when considering $\mathcal{IIS}$, it is not enough for processes not to crash for them to be able to communicate with the other processes. When considering a correct process in $\mathcal{IIS}$, the values it writes in the memory are accessible to any other process that reads the memory later in the execution. In contrast, in $\mathcal{IIS}$, all the communication occurs through the immediate snapshot objects associated with each round. It entails that the values written by a slow but correct process p_i may never be accessible to faster processes that would always access the object strictly before p_i. This has an impact on the ability for a process like p_i to take a leading role in such an execution, even if it never crashes.

To address this particularity of the $\mathcal{IIS}$ model, it is useful to introduce the set of strongly correct processes. Informally, a strongly correct process is a process whose writes in the immediate snapshot object are seen infinitely often, directly or not, by all the correct processes. In the following, a correct but not strongly correct process is called *weakly correct*.

Formal definition Let $view_j[r]$ be the view obtained by p_j at round r. Let $\mathcal{SC}_0$ be the set defined as follows (let us remember that $\mathcal{C}$ denotes the set of correct processes):

$$\mathcal{SC}_0 \stackrel{def}{=} \{\ i \text{ such that } |\{r \text{ such that } \forall\ j \in \mathcal{C} :\ \exists\ \langle i, - \rangle \in view_j[r]\}| = \infty\ \},$$

i.e., $\mathcal{SC}_0$ is the set of processes that have issued an infinite sequence of (not necessarily consecutive) invocations of write_snapshot() and these invocations have been seen by each correct process (this is because these invocations are set-linearized in the first position when considering the corresponding one-shot immediate snapshot objects).

Let us observe that, as soon as we assume that there is at least one correct process, it follows from the fact that the number of processes is bounded that $|\mathcal{SC}_0| \neq 0$. Given $k > 0$, let us recursively define $\mathcal{SC}_k$ as follows:

$$\mathcal{SC}_k \stackrel{def}{=} \{\ i \text{ such that } |\{r \text{ such that } \exists\ j \in \mathcal{SC}_{k-1} :\ \exists\ \langle i, - \rangle \in view_j[r]\}| = \infty\ \}.$$

Hence, $\mathcal{SC}_k$ contains all the correct processes that have issued an infinite sequence of (not necessarily consecutive) invocations of write_snapshot() which have been seen by at least one process of $\mathcal{SC}_{k-1}$. It follows from the self-inclusion property of the views and the definition of $\mathcal{SC}_k$ that $\mathcal{SC}_0 \subseteq \mathcal{SC}_1 \subseteq \cdots$. Moreover, as all the sets are included in $\{1, \ldots, n\}$, there is some K such that $\mathcal{SC}_0 \subseteq \mathcal{SC}_1 \subseteq \cdots \subseteq \mathcal{SC}_K = \mathcal{SC}_{K+1} = \mathcal{SC}_{K+2} = \cdots$.

$\mathcal{SC}_K$ defines the set of of strongly correct processes which is denoted $\mathcal{SC}$. This is the set of processes that have issued an infinite sequence of (not necessarily consecutive) invocations of write_snapshot() which have been propagated to all the correct processes.

2.2.2 Defining Failure Detectors for $\mathcal{IIS}$

The proposed generic approach for designing failure detectors for $\mathcal{IIS}$ relies on the notion of strongly correct processes. When considering the definition of a failure detector C in $\mathcal{IIS}$, replacing any occurrence of the word "correct" by "strongly correct" provides a failure detector C^* for $\mathcal{IIS}$. Moreover, it is assumed that during an execution in $\mathcal{IIS}[C^*]$ (the $\mathcal{IIS}$ model enriched with the failure detector C^*) at the beginning of each round, a process p_i reads and saves the value provided by the failure detector before accessing the immediate snapshot object of the round.

This way of defining failure detectors for $\mathcal{IIS}$ keeps the clean modular semantics of failure detectors while providing meaningful information in the constrained framework of $\mathcal{IIS}$. Moreover the transformation needed to obtain failure detectors suited for $\mathcal{IIS}$ from those well studied, designed for $\mathcal{ARW}$, is completely syntactic and only relies on the notion of strongly correct processes.

2.3 Simulation from $\mathcal{IIS}[C^*]$ to $\mathcal{IIS}[C]$

This section describes a simulation in $\mathcal{IIS}[C^*]$ of a run of an algorithm designed for $\mathcal{ARW}[C]$. Except for the simulation of the detector output, this simulation is from [14]. In order not to confuse a simulated process in $\mathcal{ARW}[C]$ and its simulator in $\mathcal{IIS}[C^*]$, the first one is denoted p_i while the second one is denoted q_i.

2.3.1 Description of the simulation

It is assumed, without loss of generality, that the simulated processes communicate through a single snapshot object S. A simulator q_i is associated with each simulated process p_i. It locally executes the code of p_i and uses the algorithms described in Figure 2.1 when p_i invokes S.write$(-)$, S.snapshot() or queries the failure detector.

Immediate snapshot objects of $\mathcal{IIS}[C^*]$ These objects are denoted $IS[1]$, $IS[2]$, ... Each object $IS[r]$ stores a set of triples (this set is denoted ops_i in Figure 2.1). If the triple $(j, sn, x) \in ops_i$, then the simulator q_i knows that the process p_j (simulated by q_j) has issued its sn-th invocation of an operation on the simulated snapshot object S; $x \neq \perp$ means that this invocation is S.write(x) while $x = \perp$ means that it is S.snapshot().

Local variables of a simulator q_i The variable r_i contains the current round number of the simulator q_i. It is increased before each invocation of $IS[r_i]$.write_snapshot(ops_i) (line 3). As this is the only place where, during a round, a simulator invokes the operation write_snapshot(), the simulators obey the $\mathcal{IIS}$ model.

The local variable sn_i is a sequence number that measures the progress of the simulation by q_i of the process p_i. It is increased at line 1 when p_i starts simulating a new invocation of S.write() or S.snapshot() on behalf of p_i.

As already indicated, the local variable ops_i contains the triples associated with all the invocations of S.write() and S.snapshot() that have been issued by the processes and are currently known by the simulator q_i. This local variable (which can only grow) is updated at line 1 when q_i starts simulating the next operation S.write() or S.snapshot() issued by p_i or at line 4 when q_i learns operations on the snapshot object S issued by other processes.

The local variable iis_view_i stores the value returned by the last invocation of the operation $IS[r_i]$.write_snapshot() issued by the simulator q_i (line 3). When simulating an invocation of S.snapshot() issued by p_i, q_i computes for each simulated process p_j the sequence number max_snj_i (line 12) of the last value it knows (saved in v_snj_i at line 13) that has been written by p_j in the snapshot object S. This allows q_i to compute the view arw_view_i (line 14) that it returns (line 17) as the result of the invocation of S.snapshot() issued by p_i.

The local variable fd_i is used to store the last value obtained by the simulator q_i from its read-only local failure detector variable denoted C^*.read().

Simulation of S.write(v) To simulate the invocation of S.write(v) issued by p_i, the simulator q_i invokes the internal operation publicize&progress(v). It first increments sn_i and adds the triple (i, sn_i, v) to ops_i (line 1). Then, it repeatedly invokes write_snapshot(ops_i) on successive immediate snapshot objects and enriches its set of triples ops_i (lines 2-4) until it obtains a view iis_view_i in which all the simulators it sees in that view are aware of the invocation of the operation S.write(v) that it is simulating (line 6).

Simulation of S.snapshot() To simulate an invocation of S.snapshot() issued by p_i, the simulator q_i first invokes publicize&progress$(\perp)$. When this invocation terminates, q_i knows that all the simulators it sees in the last view iis_view_i it has obtained are aware of its invocation of S.snapshot(). Moreover, as we have seen, the execution of publicize&progress$(\perp)$ makes q_i aware of operations simulated by other simulators.

Then the simulator q_i browses all the operations it is aware of in order to extract, for each simulated process p_j, the last value effectively written by p_j (lines 10-16). This (non-$\perp$) value is

```
Init: ops_i ← ∅; r_i ← 0; sn_i ← 0; iis_view_i ← ∅; fd_i ← C*.read().

internal operation publicize&progress (x) is
(1)  sn_i ← sn_i + 1; ops_i ← ops_i ∪ {(i, sn_i, x)};
(2)  repeat r_i ← r_i + 1; fd_i ← C*.read();
(3)       iis_view_i ← IS[r_i].write_snapshot(ops_i);
(4)       ops_i ← ⋃_{⟨k,ops_k⟩∈iis_view_i} ops_k
(5)  until ((i, sn_i, x) ∈ ⋂_{⟨k,ops_k⟩∈iis_view_i} ops_k) end repeat;
(6)  return().

operation S.write(v) is
(7)  publicize&progress (v); return().

operation S.snapshot() is
(8)  publicize&progress (⊥);
(9)  arw_view_i ← ∅;
(10) for each j in {1, ..., n} do
(11)  if (∃v | (j, −, v) ⋂_{⟨k,ops_k⟩∈iis_view_i} ops_k ∧ v ≠ ⊥)
(12)    then max_snj_i ← max{sn | (j, sn, v) ∈ ⋂_{⟨k,ops_k⟩∈iis_view_i} ops_k ∧ v ≠ ⊥};
(13)         vj_i ← v such that (j, max_snj_i, v) ∈ ops_i;
(14)         arw_view_i ← arw_view_i ∪ {⟨j, vj_i⟩}
(15)  end if
(16) end for;
(17) return (arw_view_i).

operation C.read() is return (fd_i).
```

Figure 2.1: Simulation of $\mathcal{ARW}[C]$ in $\mathcal{IIS}[C^*]$: code for a simulator q_i (extended from [14])

extracted from the triple with the largest sequence number among all those that appear in all the sets ops_k that belong to the view iis_view_i returned to q_i by its last invocation of write_snapshot().

Simulation of C.read() When a process p_i reads its local failure detector output, the simulator q_i simply returns it the current value of fd_i.

2.3.2 From strongly correct simulators to correct simulated processes

Strongly correct vs weakly correct simulators Let $\mathcal{WC} = \mathcal{C} \setminus \mathcal{SC}$ (the set of weakly correct simulators). It follows from the definition of the strongly correct simulators that, for any simulated process p_i whose simulator q_i is such that $i \in \mathcal{WC}$, there is a round $rmin_i$ such that, $\forall j \in \mathcal{SC}$, $\forall r \geq rmin_i : \langle i, - \rangle \notin iis\_view_j[r]$, which means that, for $r \geq rmin_i$, no invocation $IS[r]$.write_snapshot() issued by the simulator q_i is seen by a strongly correct simulator.

This means that, after $rmax = \max\{rmin_i\}_{i \in \mathcal{WC}}$ and after all simulator crashes have occurred, the invocations of write_snapshot() by the simulators of $\mathcal{SC}$ are always set-linearized strictly before the ones of the simulators of $\mathcal{WC}$. Said differently, there is a round after which no strongly correct simulator ever receives information from a weakly correct simulator. From the point of view of a strongly correct simulator, any weakly correct simulator appears as a crashed simulator. Differently, any weakly correct simulator receives forever information from all the strongly correct simulators[1].

[1]This situation is similar to the case where, in the base read/write model, after some finite time, some subset of processes (the ones corresponding here to the set of weakly correct simulators) commit forever send omission failures with respect to the correct processes (the ones corresponding here to the set of strongly correct simulators).

Crashed and slow $\mathcal{IIS}$ simulators simulate crashed $\mathcal{ARW}$ processes An important feature of the simulation described in Figure 2.1 is that, not only the crash of a simulator q_i gives rise to the crash of the associated simulated process p_i, but a slow simulator q_j entails the crash of its simulated process p_j.

As an example, let us consider a correct simulator q_j which, at any round r, is always strictly the last simulator which invokes $IS[r]$.write_snapshot(). It follows that no other simulator is ever informed of the operations S.write() and S.snapshot() issued by the process p_j simulated by q_j. When the simulator q_j executes line 3, it is informed of the operations issued by the strongly correct simulators q_i on behalf of the processes they simulate but, as the operation of p_j it is simulating (which is encoded in the triple (j, sn_j, x)) is never known by the other simulators, it follows that q_j loops forever in the **repeat** loop (lines 2-5). Hence, the simulation of p_j does not longer progress and p_j is considered as a crashed process in the simulated $\mathcal{ARW}$ model. This means that the simulation described in Figure 2.1 guarantees wait-freedom for the processes simulated by the strongly correct simulators only.

To summarize When simulating $\mathcal{ARW}[C]$ on top of $\mathcal{IIS}[C^*]$, we have the following: (a) a faulty or weakly correct simulator q_i gives rise to a faulty simulated process p_i and (b) a strongly correct process gives rise to a correct simulated process p_i in $\mathcal{ARW}[C]$.

The next theorem captures the previous discussion.

Theorem 1 *Let A be an algorithm solving a colorless task in the $\mathcal{ARW}[C]$ model. Let us consider an execution of A simulated in the $\mathcal{IIS}[C^*]$ model by the algorithms S.write(), S.snapshot() and C.read() described in Figure 2.1. A process p_i is correct in the simulated execution if and only if its simulator q_i is strongly correct in the simulation.*

Proof Let us first observe that a simulated process whose simulator eventually crashes in $\mathcal{IIS}_n$ is faulty (it cannot issue an infinite number of steps since its simulator crashes after a finite number of steps). Hence, it remains to show that (a) all simulators in $\mathcal{WC}$ simulate faulty processes and (b) all simulators in $\mathcal{SC}$ simulate correct processes.

Proof of (a). Let us first remark that, as there is a bounded number of simulators and (by assumption) at least one of them is correct, $|\mathcal{SC}| \geq 1$. If $\mathcal{C} = \mathcal{SC}$ (i.e., $\mathcal{WC} = \emptyset$), (a) trivially follows. Hence, let us consider that there is at least one weakly correct simulator q_i.

Let $rmax$ be a round number such that (1) $\forall i \in \mathcal{WC}, \forall j \in \mathcal{SC}, \forall r \geq rmax : \langle i, - \rangle \notin is\_view_j[r]$ and (2) no faulty simulator starts round $rmax$. At any round $r \geq rmax$, any strongly correct simulator issues its invocation of $IS[r]$.write_snapshot() strictly before those of any weakly correct simulator. Consequently, when after round $rmax$, q_i, $i \in \mathcal{WC}$, executes the algorithm that simulates the next invocation of S.write() or S.snapshot() issued by p_i, encoded in the triple (i, sn, x), the only simulators that can see this triple are weakly correct simulators. Then, as after $rmax$ the simulator q_i sees all the triples written or known by the strongly correct simulators, its predicate of line 5 will never be verified and q_i will loop forever. Hence the simulated process p_i never ends its invocation of S.write() or S.snapshot(): it does no longer progress and appears as a crashed process in the simulated execution in $\mathcal{ASW}_n[C]$.

Proof of (b). Let now q_i be a strongly correct simulator. By definition of $\mathcal{SC}$, there is a set $\mathcal{SC}_k$ such that $i \in \mathcal{SC}_k$ and there is an infinite sequence of invocations of $IS[r]$.write_snapshot() issued by q_i which occur before those of some simulators of $\mathcal{SC}_{k-1}$. It follows that, each time q_i simulates a S.write() or S.snapshot() invocation on behalf of p_i, after a finite number of rounds, the corresponding triple (i, sn, x) is added to the set of its known simulated operations by a simulator of $\mathcal{SC}_{k-1}$. Then, in the same way and in a finite number of rounds, that simulator propagates this

triple to a simulator $\mathcal{SC}_{k-2}$. This continues recursively and, in a finite number of rounds, the triple (i, sn, x) written by q_i is added to the set ops_y of known operations by a simulator q_y, $y \in \mathcal{SC}_0$. Then, it follows from the definition of $\mathcal{SC}_0$ that q_y propagates the triple that becomes eventually known by all correct simulators. After that, the next invocation of $S[r'].\mathsf{write\_snapshot}()$ issued by q_i is such that the predicate of line 5 is satisfied and q_i can terminate its simulation of the operation $S.\mathsf{write}()$ or $S.\mathsf{snapshot}()$ issued by the process p_i. It follows that the simulated process p_i always progresses and it is consequently correct with respect to the simulated execution in $\mathcal{ARW}_n[C]$. $\square_{Theorem\ 1}$

2.4 Simulation from $\mathcal{ARW}[C]$ to $\mathcal{IIS}[C^*]$

This section presents a generic simulation of $\mathcal{IIS}[C^*]$ in $\mathcal{ARW}[C]$. Its generic dimension lies in the fact that C can be any failure detector class among P, Σ, $\diamond P$, Ω, Ω_k and others such as S, $\diamond S$ [20] and $\diamond S_x$ [5]. As far as terminology is concerned, q_i is used to denote a simulated $\mathcal{IIS}$ process while p_i is used to denote the corresponding $\mathcal{ARW}$ simulator process. The simulation is described in Figure 2.2. Differently from the simulation described in Figure 2.1, the algorithms of Figure 2.2 are not required to be full-information algorithms.

2.4.1 Description of the simulation

The simulated model $IS[1]$, $IS[2]$, ... denotes the infinite sequence of one-shot immediate snapshot objects of the simulated $\mathcal{IIS}$ model. Hence, a simulated process q_i invokes the operations $IS[r].\mathsf{write\_snapshot}()$ and $C^*.\mathsf{read}()$.

Shared objects of the simulation The simulation uses an infinite sequence of objects $S[1]$, $S[2]$, ... The object $S[r]$ is used to implement the corresponding one-shot immediate snapshot object $IS[r]$.

Each of these objects $S[r]$ can be accessed by two operations that are denoted $\mathsf{collect}()$ and $\mathsf{arw\_write\_snapshot}()$. The latter is nothing else than the operation $\mathsf{write\_snapshot}()$ (which satisfies the self-inclusion, containment and immediacy properties defined in Section 2.1). It is prefixed by "arw" in order not to be confused with the operation of the $\mathcal{IIS}$ model that it helps simulate. The operation $\mathsf{collect}()$ is similar to the operation $\mathsf{snapshot}()$, except that it is not required to be atomic. It consists in an asynchronous scan of the corresponding $S[r]$ object which returns the set of pairs it has seen in $S[r]$. Both $\mathsf{collect}()$ and $\mathsf{arw\_write\_snapshot}()$ can be wait-free implemented in $\mathcal{ARW}[\emptyset]$.

FD_VAL is an array of single-writer/multi-reader atomic registers. The simulator p_i stores in the register $FD\_VAL[i]$ the last value it has read from its local failure detector variable which is denoted $C.\mathsf{read}()$.

Where is the problem to solve If the underlying model was $\mathcal{ARW}[\emptyset]$ (no failure detector), the simulation of the operation $IS[r].\mathsf{write\_snapshot}()$ would boil down to a simple call to $S[r].\mathsf{arw\_write\_snapshot}()$ (lines 6-7). Hence, the main difficulty to overcome in order to simulate $IS[r].\mathsf{write\_snapshot}(v)$ comes from the presence of the failure detector C.

This comes from the fact that, in all executions, we need to guarantee a correct association between the schedule of the (simulated) invocations of $IS[r].\mathsf{write\_snapshot}()$ and the outputs of the simulated failure detector C^*. This, which depends on the output of the underlying failure detector C, requires to appropriately synchronize, at every round r, the simulation of the invocations of $IS[r].\mathsf{write\_snapshot}()$. Once, this is done, the set-linearization of the simulated invocations

```
operation IS[r].write_snapshot(v) is
(1)  if ((r mod n) + 1 = i)
(2)    then repeat arw_view_i ← S[r].collect(); FD_VAL[i] ← C.read()
(3)         until (PROP_C(arw_view_i)) end repeat
(4)    else FD_VAL[i] ← C.read()
(5)  end if;
(6)  iis_view ← S[r].arw_write_snapshot(v);
(7)  return (iis_view).

operation C*.read() is return (FD_VAL[i]).
```

Figure 2.2: A generic simulation of $\mathcal{IIS}[C^*]$ in $\mathcal{ARW}[C]$: code for a simulator p_i

of $IS[r]$.write_snapshot() follows from the set-linearization of these invocations in the $\mathcal{ARW}[C]$ model.

Associate each round with a simulator The simulation associates each round r with a simulator (we say that the corresponding simulator "owns" round r) in such a way that each correct simulator owns an infinite number of rounds. This is implemented with a simple round-robin technique (line 1).

Simulation of $IS[r]$.write_snapshot(v) To simulate an invocation $IS[r]$.write_snapshot(v) issued by the simulated process q_i, the simulator p_i first checks if it is the owner of the corresponding round r. If it is not, it refreshes the value of $FD\_VAL[i]$ (line 4) and executes the "common part", namely, it invokes $S[r]$.arw_write_snapshot(v) (line 6) which returns it a set iis_view that constitutes the result of the invocation of $IS[r]$.write_snapshot(v).

If the simulator p_i is the owner of the round, it repeatedly reads asynchronously the current value of the implementation object $S[r]$ (that it stores in arw_view_i) and refreshes the value of $FD\_VAL[i]$ (line 2). This **repeat** statement terminates when the values of arw_view_i it has obtained satisfy some predicate (line 3). This predicate, denoted PROP$_C()$, which depends on the failure detector class C, encapsulates the generic dimension of the simulation. Then, after it has exited the loop, the simulator p_i executes the "common" part, i.e., lines 6-7. It invokes $S[r]$.arw_write_snapshot(v) which provides it with a view iis_view which is returned as the result of the invocation of $IS[r]$.write_snapshot(v).

The fact that, during each round, (a) some code is executed only by the simulator that owns r, (b) some code is executed only by the other simulators and (c) some code is executed by all simulators, realizes the synchronization discussed above that allows for a correct set-linearization of the invocations of $IS[r]$.write_snapshot() in $\mathcal{IIS}[C^*]$.

Simulation of C^*.read() When a simulated process q_i wants to read its local failure detector output, its simulator p_i returns it the last value it has read from its local failure detector variable.

To summarize When simulating $\mathcal{IIS}[C^*]$ on top of $\mathcal{ARW}[C]$, we have the following: (a) a faulty simulator p_i gives rise to a faulty simulated process q_i and (b) a correct simulator p_i gives rise either to a strongly correct, a weakly correct or a faulty simulated process q_i in $\mathcal{IIS}[C^*]$ (this can depend on PROP$_C()$).

Moreover, whatever C, we have to show that there is at least one correct process in $\mathcal{IIS}[C^*]$. This amounts to show that there is a simulator p_i that executes the infinite sequence of operations $\{IS[r]$.write_snapshot$()\}_{r\geq 1}$. To that end, we have to show that each object $IS[r]$ is non-blocking [38] (i.e., whatever the round r and the concurrency pattern, at least one invocation of

$IS[r]$.write_snapshot() terminates). The corresponding proof is given when we consider specific failure detector classes (see below). Then, due to the structure of the $\mathcal{IIS}$ model, the very existence of at least one correct process in $\mathcal{IIS}[C^*]$ entails the existence of at least one strongly correct process in this model (see the definition of the set $\mathcal{SC}$ in Section 2.2.1).

2.4.2 Instantiating the simulation with $C = \Omega_k$

The failure detector class $C = \Omega_k$ The failure detector Ω_k introduced in [70] is a straightforward generalization of the failure detector class Ω introduced in [19]. It provides each process p_i with a read-only local variable $leaders_i$ that always contains k process indexes and satisfies the following property.

- The local variables $leaders_i$ offered by a failure detector of the class Ω_k are such that, after some unknown but finite time τ, they all contain forever the same set of k process indexes and at least one of these indexes is the one of a correct process.

Let us notice that, before time τ, the sets $leaders_i$ can contain arbitrarily changing sets of k process indexes.

The property PROP_{Ω_k} When $C = \Omega_k$, the property $\text{PROP}_C(arw\_view)$ can be instantiated at each simulator p_i as follows:

$$\text{PROP}_{\Omega_k}(arw\_view_i) \;=\; \big(\exists\, \ell \in FD\_VAL[i] :\; (\ell = i \;\vee\; \exists \langle \ell, - \rangle \in arw\_view_i)\big).$$

Let $leaders_i = FD\_VAL[i]$ (the last value of Ω_k read by the simulator p_i). The previous predicate directs the simulator p_i, at each round r it owns, to wait until $i \in leaders_i$ or until it has seen the simulation of $IS[r]$.write_snapshot() issued by a simulator q_j such that $j \in leaders_i$.

Theorem 2 *Let A be an algorithm solving a colorless task in the $\mathcal{IIS}[\Omega_k^*]$ model. The simulation of A on top of $\mathcal{ARW}[\Omega_k]$ where the invocations of $IS[r]$.write_snapshot() and C^*.read() are implemented by the algorithms described in Figure 2.2 and the predicate PROP_C is instantiated by PROP_{Ω_k}, produces an execution of A that could have been obtained in $\mathcal{IIS}[\Omega_k^*]$. Moreover, there is a one-to-one correspondence between the correct (simulated) processes in $\mathcal{IIS}[\Omega_k^*]$ and the correct simulators in $\mathcal{ARW}[\Omega_k]$.*

Proof The proof has to show that: (a) there is at least one correct process in $\mathcal{IIS}[\Omega_k^*]$ (consequently, as we have seen previously, there is a strongly correct process in $\mathcal{IIS}[\Omega_k^*]$); (b) there is a one-to-one correspondence between correct simulators (p_i) and correct simulated processes (q_i); (c) the behavior of the local failure detector variables of the processes in $\mathcal{IIS}[\Omega_k^*]$ is the one defined by Ω_k^*; and, for any round r, (d) the invocations of $IS[r]$.write_snapshot() satisfy the self-inclusion, containment, and immediacy properties (defined in Section 2.1).

Proof of (a). As (by definition) one of the leaders eventually output by Ω_k is a correct simulator p_i, there is a finite time τ after which the predicate PROP_{Ω_k} returns always *true* when evaluated by the simulator p_i. Hence, this simulator can never be blocked forever at line 3 when it executes (on behalf of the simulated process q_i) the algorithm implementing the operation $IS[r]$.write_snapshot(). It follows from this observation that there is at least one correct simulated process q_i.

Proof of (b). The correct simulator p_i which eventually belongs forever to the output of Ω_k invokes arw_write_snapshot(v) at each simulated round r (line 6). Let τ_r be the finite time instant

at which this invocation terminates. As there is a finite time $\tau' \geq \tau$ after which i belongs to all the failure detector outputs, it follows that, at some finite time $\tau'' \geq \max(\tau', \tau_r)$, the evaluation of PROP_{Ω_k} by any correct simulator returns $true$ at round r. Hence any correct simulator terminates its simulation of $IS[r]$.write_snapshot(). As this is true for any round r, any correct simulator simulates an infinite number of rounds of the $\mathcal{IIS}$ model. Consequently, if a simulator p_j is correct in $\mathcal{ARW}[\Omega_k^*]$ the associated simulated process q_j is correct in $\mathcal{IIS}[\Omega_k^*]$ (which entails that, at any round r, the simulation of the operation $IS[r]$.write_snapshot() is wait-free). Finally, a faulty simulator p_i trivially gives rise to a faulty simulated process q_i which concludes the proof of Item (b).

Proof of (c). To show that the behavior of the local failure detector variables at each simulated process q_x is the one defined by Ω_k^*, let us first observe that, it follows directly from lines 2 and 4 that the outputs of Ω_k^* are outputs of Ω_k. Hence, we have only to show that, after some time, Ω_k outputs forever a set L such that there is a simulator p_i with $i \in L \cap \mathcal{C}$ (let us remember that $\mathcal{C}$ is the set of correct simulators) and the simulated process q_i associated with the simulator p_i is strongly correct.

Let r be a round such that all the faulty simulators have crashed before r and, after r, all correct simulators obtain forever the same set of leaders L from Ω_k. Due to PROP_{Ω_k}, in all rounds $r' \geq r$, each correct simulator p_j, $j \notin L$, has to wait (at each round it owns) until a correct simulator p_i, $i \in L$, has written into $S[r']$ (execution of $S[r']$.arw_write_snapshot() by p_i at line 6 and execution of $S[r']$.collect() by p_j at line 2). It follows that, in the simulated system, the invocation of $IS[r']$.write_snapshot() issued by any simulated process q_j, $j \notin L$, is set-linearized after the invocation of $IS[r']$.write_snapshot() issued by a simulated process p_i such that $i \in L$.

Let q_j be a strongly correct simulated process (since there are some correct simulated processes, there is at least one strongly correct one). As q_j executes an infinite number of rounds and $1 \leq |L \cap \mathcal{C}| \leq k$, it follows that there is a correct simulated process q_ℓ such that $\ell \in L \cap \mathcal{C}$ and there is an infinite number of rounds r'' such that the invocation of $IS[r'']$.write_snapshot() issued by q_j is set-linearized after the one of q_ℓ. It follows that q_ℓ is a (simulated process which) is strongly correct.[2]

Proof of (d). The fact that, at any round r, the invocations of $IS[r]$.write_snapshot() satisfy the self-inclusion, containment and immediacy properties follow directly (as already indicated) from the fact that invocations of the underlying operation $S[r]$.arw_write_snapshot() satisfy these properties. $\square_{Theorem\ 2}$

The following corollary is an immediate consequence of the previous theorem.

Corollary 1 *A colorless task T is solvable in $\mathcal{ARW}[\Omega_k]$ iff it is solvable in $\mathcal{IIS}[\Omega_k^*]$.*

2.4.3 Instantiating the simulation with $C = \diamond P$

The failure detector class $C = P$ and $C = \diamond P$ The perfect failure detector P and the eventually perfect failure detector $\diamond P$ have been defined in [20]. They verify the following properties.

- A failure detector of the class P provides each process p_i with a set $trusted_i$ that, at any time τ, contains all the processes that have not crashed by time τ and eventually contains

[2] Let us observe that, while this proves that there is a correct simulator that gives rise to a strongly correct simulated process, we cannot determine how many simulated processes are strongly correct. This number depends on the execution.

only correct processes. [3]

- The class $\diamond P$ is weaker than the class P. Namely, there is an arbitrary long finite period during which the sets $trusted_i$ can contain arbitrary values and when this period terminates a failure detector of $\diamond P$ behaves as one of class P.

The property PROP$_{\diamond P}$ When $C = \diamond P$, the property PROP$_{\diamond P}(arw_view)$ can be instantiated at each simulator p_i as follows:

$$\text{PROP}_{\diamond P}(arw\_view_i) \;=\; \big(\forall\, j \in FD\_VAL[i] :\; (j = i \;\vee\; \langle j, -\rangle \in arw\_view_i)\big).$$

This property forces the corresponding simulator p_i, at each round r it owns, to wait until all the simulators p_j that it currently trusts (i.e., any $j \in fd\_val_i(i)$) have invoked the operation $S[r]$.arw_write_snapshot() (i.e., have written a pair $\langle j, -\rangle$ into $S[r]$).

Theorem 3 *Let A be an algorithm solving a colorless task in the $\mathcal{IIS}[\diamond P^*]$ model. The simulation of A on top of $\mathcal{ARW}[\diamond P]$ where the invocations of $IS[r]$.write_snapshot() and $\diamond P^*$.read() are implemented by the algorithms described in Figure 2.2 and the predicate* PROP$_C$ *is instantiated by* PROP$_{\diamond P}$*, produces an execution of A that could have been obtained in $\mathcal{IIS}[\diamond P^*]$. Moreover, there is a one-to-one correspondence between the correct (simulated) processes in $\mathcal{IIS}[\diamond P^*]$ and the correct simulators in $\mathcal{ARW}[\diamond P]$ and all correct simulated processes are strongly correct.*

Proof The proof is similar to the previous one. We show here only that each correct simulator p_i gives rise to a strongly correct (simulated) process q_i.

The proof is by contradiction. Let us suppose that a correct simulator loops forever in the **repeat** (lines 2-3) when it executes the algorithm simulating $IS[r]$.write_snapshot(). Let r denote the first round during which this happens and let p_i be the simulator that owns this round (hence, p_i is the first simulator that loops forever).

Let us notice that, as $i = (r \mod n) + 1$, it is the only simulator which loops at round r. Hence, all other correct simulators eventually invoke $S[r]$.arw_write_snapshot() at line 6 and consequently return from their simulation of $IS[r]$.write_snapshot().

As the failure detector $\diamond P$ eventually outputs a set including only correct simulators, the evaluation of the predicate PROP$_{\diamond P}(r, -, -)$ eventually returns *true* to p_i which terminates its simulation of $IS[r]$.write_snapshot(). A contradiction. It follows from this contradiction that every correct simulator executes an infinite number of rounds, which means that, at any round r, the implementation of the operation $IS[r]$.write_snapshot() is wait-free.

Finally, as, at each round it owns, each correct simulator p_i waits for all other correct simulators, each correct simulator sees the simulation of $IS[r]$.write_snapshot() by the other correct simulators infinitely often. Hence, each correct simulator p_i simulates a strongly correct process q_i. $\square_{Theorem\ 3}$

2.4.4 Instantiating the simulation with $C = P,\ \Sigma,\ S,\ \diamond S,\ S_x,\ \diamond S_x$

The same predicate PROP$_C$ as the one used for $\diamond P$ works for these failure detector classes. The definitions of these failure detector classes can be found in Subsection 2.4.3 for P, in [25] for Σ, in [64] for S and $\diamond S$ and in [5] for S_x and $\diamond S_x$.

[3] The traditional definition of P provides each process p_i with a set $faulty_i$ that does not contain a process before it crashes and eventually contains all faulty processes. It is easy to see that $trusted_i = \Pi \setminus faulty_i$.

The failure detector classes $C = P,\ \Sigma,\ S,\ \diamond S$ The previous proof can be easily translated for the failure detector classes $C = P,\ \Sigma,\ S,\ \diamond S$. This follows from the observation that, as $\diamond P$, each of these failure detector classes permanently ($C = P$) or eventually ($C = \Sigma,\ S,\ \diamond S$) output only sets of correct processes.

The failure detector classes $C = S_x,\ \diamond S_x$ These failure detector classes extend the classes S and $\diamond S$. Intuitively, they restrict the properties defining S and $\diamond S$ to be only on a dynamically determined subset of processes Q such that $|Q| = x$ (hence their name: limited scope failure detector classes).

It is possible to show that the algorithm of Figure 2.2 instantiated with the previous predicate $\text{PROP}_{\diamond P}()$ allows for a correct simulation of $\mathcal{IIS}[C^*]$ in $\mathcal{ARW}[C]$ for $C = S_x,\ \diamond S_x$. Let us also remark that, for any r, the simulation of the operation $IS[r]$.write_snapshot() is wait-free (each simulated process q_i whose simulator p_i is correct executes an infinite number of $\mathcal{IIS}$ rounds). However, while each correct simulator simulates a strongly correct process when $C \in \{P, \diamond P\}$, no conclusion can be drawn from the number of strongly correct simulated processes (except that there is at least one) when $C \in \{\Sigma, S_x, \diamond S_x\}$.

2.5 From wait-freedom to t-resilience

Notation Let $\mathcal{IIS}_{n,t}[C]$ denote the extended $\mathcal{IIS}[C]$ model in which at least $n - t$ processes are strongly correct, i.e., $|\mathcal{SC}| \geq n - t$ and $|\mathcal{WC}| + |\mathcal{F}| \leq t$. Similarly, let $\mathcal{ARW}_{n,t}[C]$ denote the extended $\mathcal{ARW}[C]$ model in which at most t processes are faulty.

From $\mathcal{IIS}_{n,t}[C^*]$ to $\mathcal{ARW}_{n,t}[C]$ Theorem 1 has shown that the simulation described in Figure 2.1 (which is a simple extension to failure detectors of the simulation described in [14]) ensures that (a) any strongly correct simulator in $\mathcal{IIS}$ gives rise to a correct simulated process in $\mathcal{ARW}$ and (b) a weakly or faulty simulator gives rise to a faulty simulated process. It follows that if $|\mathcal{SC}| \geq n-t$ in $\mathcal{IIS}_{n,t}[C^*]$ we have at most t faulty process in the simulated system $\mathcal{ARW}_{n,t}[C]$.

From $\mathcal{ARW}_{n,t}[C]$ to $\mathcal{IIS}_{n,t}[C^*]$ In this direction, the simulation from $\mathcal{ARW}[C]$ in $\mathcal{IIS}[C^*]$ presented in Figure 2.2 can be easily adapted in order to simulate $\mathcal{ARW}_{n,t}[C]$ in $\mathcal{IIS}_{n,t}[C^*]$.

It is indeed sufficient to replace PROP_C by $\text{PROP}_C \wedge |arw\_view_i| \geq (n-t-1)$ (it is of course assumed that we do not have $|arw\_view_i| \geq (n - t - 1) \Rightarrow \neg\ \text{PROP}_C$). In this way, at every round r it owns, each correct simulator p_i is constrained to wait until at least $n - t - 1$ processes have invoked $S[r]$.arw_write_snapshot() before being allowed to invoke its own. The correction of this extended simulation is captured in the following theorem.

Theorem 4 *Let A be an algorithm solving a colorless task in the $\mathcal{IIS}[C^*]$ model. For the failure detector classes studied in this section, the simulation of A on top of $\mathcal{ARW}[C]$ where the invocations of $IS[r]$.write_snapshot() and C^*.read() are implemented by the algorithms described in Figure 2.2 and the predicate PROP_C is replaced by $\text{PROP}_C \wedge |arw\_view_i| \geq (n - t - 1)$, produces a correct execution of A in $\mathcal{IIS}[C^*]$ in which $n - t$ processes are strongly correct.*

Proof The proof consists in showing that (a) each correct simulator that would simulate an infinite number of $\mathcal{IIS}$ rounds when considering only PROP_C does simulate an infinite number of rounds when considering $\text{PROP}_C \wedge (arw\_view_i \geq n - t - 1)$; and (b) there are at least $(n - t)$ strongly correct processes in the simulated $\mathcal{IIS}$ model.

Proof of (a). The proof is by contradiction. Let us suppose that a correct simulator p_i that, at some round r, blocks forever because the predicate $(arw\_view_i \geq n-t-1)$ is never satisfied. Let r be the first round at which this occurs. By assumption, at least $n-t-1$ simulators p_j eventually invoke $S[r]$.arw_write_snapshot(), it follows that the predicate $(arw\_view_i \geq n-t-1)$ eventually becomes true. Hence the predicate $\text{PROP}_C \wedge (arw\_view_i \geq n-t-1)$ eventually becomes true which concludes the proof of item (a).

Proof of (b). Due to the previous item, there is a correct simulator p_i that simulates a strongly correct process. Let p_i such a simulator. This simulator p_i simulates an infinite number of rounds and, in each round r it owns, it simulates $IS[r]$.write_snapshot() on behalf of q_i, after (or simultaneously with) the invocations of $IS[r]$.write_snapshot() issued by at least $n-t$ processes ($n-t-1$ other processes plus itself). As there is a bounded number of processes, p_i consequently simulates its write-snapshots infinitely often after (or simultaneously with) those of at least $n-t$ simulators. Hence at least $n-t$ simulated processes are strongly correct and (b) follows. $\square_{Theorem\ 4}$

Failure detectors in $\mathcal{IIS}$

As shown in until now in this chapter, for several failure detectors designed for $\mathcal{ARW}$, it is possible to use a syntactic transformation of their definition that allows us to use them meaningfully in $\mathcal{IIS}$. This transformation relies on the notion of strongly correct processes, that capture the ability for a process to convey information to all correct processes in the $\mathcal{IIS}$ model. The simulations used to show that the transformation preserves the computability power with respect to colorless tasks as well as their proofs are also generic and only need to be slightly modified to fit other failure detectors.

We hope that this work may be used to prove impossibility results in $\mathcal{ARW}$ enriched with a failure detector. One can indeed prove that a given colorless task is impossible to solve in the more constrained and easier to study $\mathcal{IIS}[C^*]$ model, it would then imply that it is also impossible to solve in the $\mathcal{ARW}[C]$ model.

2.6 Message Adversaries Corresponding to some Failure Detectors

In the following sections we consider a different computation model introduced in [2] that consider a system of failure-free synchronous processes. It aims at capturing the uncertainty caused by crashes and asynchrony by the use of an adversary that destroys some of the messages. The goal of this work, published in [85] is to find the adversaries that weaken the failure-free synchronous message-passing model to get models that are computationally equivalent to the asynchronous, shared memory or message-passing models enriched with some well studied failure-detectors.

Message adversaries for synchronous message-passing systems In a round-based message-passing synchronous system, processes communicate by exchanging messages at every round, and the synchrony assumption provided by the model guarantees that the messages sent at the beginning of a round are received by their destination processes by the end of the corresponding round. Assuming that no process is faulty, the notion of a *message adversary* has been introduced in [91] (where it is called *mobile transmission failures*) to model message losses and study their impact on the computability power of synchronous systems [91, 90].

Interestingly, the notion of constraining message deliveries has also been investigated in asynchronous systems, under distinct names, and in different contexts. As an example, asynchronous message patterns which allow failure detectors to be implemented despite asynchrony have been

investigated in [7, 62]. The view of failure detectors as being schedulers which encapsulate fairness assumptions can also be related to this approach [23, 72]. Recently, assumptions on message deliveries and message exchange patterns have been used to define new asynchronous computation models and study their computability power [21, 50, 74, 92]. The general idea, which underlies these works, consists in capturing the "weakest pattern of information exchange" that allows a family of problems to be solved despite failures.

Notations Let FD denote a failure detector. To avoid confusion with the message adversaries in the following sections, $\mathcal{ARW}[fd : \mathrm{FD}]$ denotes the asynchronous read/write model where up to $(n-1)$ processes may crash, enriched with FD. Similarly, $\mathcal{AMP}[fd : \mathrm{FD}]$ denotes $\mathcal{AMP}$ enriched with FD.

The notation $\mathcal{SMP}[adv : \mathrm{AD}]$ is used to denote a round-based synchronous system made up of n reliable sequential processes whose communications are under the control of the adversary AD. While, in every round, each process sends a message to each other process, the power of the adversary AD consists in suppressing some of these messages (which are consequently never received).

According to their power, several classes of adversaries can be defined. $\mathcal{SMP}$ denotes a synchronous system in which the adversary has no power (it can suppress no message), while $\mathcal{SMP}[adv : \infty]$ denotes the synchronous system in which the adversary can suppress all messages. It is easy to see that, from a message adversary and computability point of view, $\mathcal{SMP}$ is the most powerful crash-free synchronous system, while $\mathcal{SMP}[adv : \infty]$ is the weakest. More generally, the weaker the message adversary AD, the more powerful the system.

Asynchrony from synchrony Afek and Gafni addressed recently task solvability in synchronous message-passing systems weakened by message adversaries [2]. Let $\mathcal{ARW}$ denote the asynchronous read/write model where up to $(n-1)$ processes may crash. Afek and Gafni's main results are the following ones.

- Their first result concerns the adversary TOUR (for tournament) whose behavior is the following one. For each pair of processes p_i and p_j, and in each synchronous round, TOUR is allowed to suppress either the message sent by p_i to p_j or the message sent by p_j to p_i, but not both. The important result attached to TOUR is that $\mathcal{SMP}[adv : \mathrm{TOUR}]$ and $\mathcal{ARW}$ have the same computability power for read/write wait-free solvable tasks.

- In addition to TOUR, two more adversaries, denoted TP and PAIRS, are described and it is shown that the three considered adversary-based synchronous models $\mathcal{SMP}[adv : \mathrm{TOUR}]$, $\mathcal{SMP}[adv : \mathrm{TP}]$, and $\mathcal{SMP}[adv : \mathrm{PAIRS}]$ are equivalent for task solvability. Moreover, $\mathcal{SMP}[adv : \mathrm{PAIRS}]$ is used to show that, from a topology point of view, the protocol complex of PAIRS is a subdivided complex. This means that the message adversary PAIRS (and consequently also TOUR and TP) captures, in a very simple way, Herlihy and Shavit's condition equating the read/write wait-free model with a complex subdivision [45].

2.6.1 Message Adversary, Message Graphs, and Dynamic Graphs

Message adversary Given a run of a synchronous system, a message adversary suppresses messages sent by processes. A property associated with a message adversary restricts its power by specifying messages which cannot be suppressed. A message adversary is consequently defined by a set of properties which constrain its behavior.

Message graphs associated with each round of a synchronous system Given a message adversary AD, and a round r of a run of a synchronous system, let $\mathcal{G}^r$ be the directed graph (as defined in [2]), whose vertices are the process identities, and such that there is an edge from i to j iff the adversary AD does not suppress the message sent by p_i to p_j at round r. We consider the following definition associated with each graph $\mathcal{G}^r$.

- $i \stackrel{r}{\longrightarrow} j$ means that the directed edge (i, j) belongs to $\mathcal{G}^r$ (at round r, the message from p_i to p_j is not removed by the adversary).

The property TOUR As indicated earlier, the property TOUR [2] restricts the behavior of a message adversary as follows. For any r, and any pair of processes (p_i, p_j), $\mathcal{G}^r$ contains the directed edge (i, j) or the directed edge (j, i) or both. This means that, at every round, the adversary cannot suppress both the messages sent to each other by two processes. Hence, the graphs $\mathcal{G}^r$ associated with the rounds r of a run in $\mathcal{SMP}[adv : \mathrm{TOUR}]$ are such that:

$$\forall r \geq 1 : \forall\, (i, j) : (i \stackrel{r}{\longrightarrow} j) \ \vee \ (j \stackrel{r}{\longrightarrow} i).$$

Strongly/weakly correct processes in a synchronous run Similarly to the case of $\mathcal{IIS}$ described in sub-section 2.2.1, the communication pattern that occurs during a given execution in $\mathcal{SMP}$ constrained by a message adversary may prevent some processes that are correct from reaching part of the system with their messages.

Again, we consider the group of processes that are infinitely often able to send a message to any process in the system. The message can be sent directly or be relayed by other processes. Formally the notion of strongly correct processes is defined as follows.

- $i \stackrel{\geq r}{\rightsquigarrow} j$ means that there is a directed path starting from p_i and leading to p_j in a dynamically defined sequence of message graphs starting at a round $\geq r$. More formally,

$$\begin{array}{l} \exists k \geq 0, \exists r_1 < \cdots < r_k, \ \exists \lambda_0, \lambda_1, \ldots, \lambda_k \in \{1, \ldots, n\} \ : \\ \qquad (r_1 \geq r) \wedge (\lambda_0 = i \wedge \lambda_k = j) \wedge (\forall m \in \{1, \ldots, k\} \ : \ \lambda_{m-1} \stackrel{r_m}{\longrightarrow} \lambda_m). \end{array}$$

- $i \stackrel{\infty}{\rightsquigarrow} j \stackrel{def}{=} \big(\forall r > 0 \ : i \stackrel{\geq r}{\rightsquigarrow} j\big)$. Hence, $i \stackrel{\infty}{\rightsquigarrow} j$ means that, whatever r, there is eventually a directed path starting at p_i at a round $\geq r$ and finishing at p_j in the dynamically defined sequence of message graphs.

- $(i \stackrel{\infty}{\leftrightsquigarrow} j) \Leftrightarrow (i \stackrel{\infty}{\rightsquigarrow} j \wedge j \stackrel{\infty}{\rightsquigarrow} i)$. Assuming each process always receives its own messages, this relation is reflexive, symmetric, and transitive. Hence, it is an equivalence relation.

- Let G be the graph whose vertices are $\{1, ..., n\}$ and directed edges are defined by the relation $\stackrel{\infty}{\rightsquigarrow}$; let $SC(G)$ be the graph of its strongly connected components. If $SC(G)$ has a single vertex X with no input edge, the processes in X are called *strongly correct* processes, while the processes in $\{1, ..., n\} \setminus X$ are called *weakly correct*. If X is not unique, all processes are weakly correct.

Let $\mathcal{SC}$ denote the (possibly empty) set of strongly correct processes in a synchronous round-based system under the control of a message adversary.

2.7 SOURCE + TOUR is a Characterization of Ω in $\mathcal{ARW}$

This section shows that the computing models $\mathcal{SMP}[adv : \mathrm{SOURCE}, \mathrm{TOUR}]$ and $\mathcal{ARW}[fd : \Omega]$ have the same computational power for tasks.

2.7.1 The Property SOURCE

This property is defined as follows:

$$\exists s \in \{1,\ldots,n\} : \exists r_0 \geq 1 : \ \forall r \geq r_0 : \ \forall\, i \in \{1,\ldots,n\} : \ (s \stackrel{r}{\longrightarrow} i).$$

This statement means that, in each run of $\mathcal{SMP}[adv : \text{SOURCE}]$, there are a process p_s and a round r_0, such that, at every round $r \geq r_0$, the adversary does not suppress the message sent by p_s to the other processes.

2.7.2 From $\mathcal{ARW}[fd : \Omega]$ to $\mathcal{SMP}[adv :$ SOURCE, TOUR$]$

This section presents a simulation of $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$ on top of $\mathcal{ARW}[fd : \Omega]$ such that, any task that can be solved in $\mathcal{ARW}[fd : \Omega]$ can be solved in $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$.

Global and local variables of the simulation The simulation uses a set of shared variables $MEM[1..n][1..][1..n]$ where $MEM[i][r][j]$ is an atomic read/write register written by p_i and read by p_j. This register contains the message sent by p_i to p_j during the round r of the simulation of $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$; $\perp$ is a default value used to indicate that no message has yet been written or the corresponding message has been suppressed by the adversary.

The local variable r_i simulates the current round number of $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$, while ls_state_i represents the local simulation state. The local variable $msgs\_to\_send_i[1..n]$ contains the messages that p_i will send to each other process during the next simulated round ($msgs\_to\_send_i[j]$ contains the message for p_j). $leader_i$ is the read-only local variable containing the current local output of Ω.

The simulation is locally defined by the function simulate() which takes as input parameters the current local state of the simulation and the messages received from the other processes at the current round. It modifies accordingly the local simulation state and computes the messages that will be sent to the other processes during the next round.

The simulation algorithm The local simulation algorithm is described in Figure 2.3. The local simulator of process p_i first proceeds to the next round (line 5) and waits until its current leader has sent it a message ($MEM[leader_i][r_i][i] \neq \perp$) or it is its own leader (lines 6-8). When this occurs, the simulator writes in $MEM[i][r_i]$ the messages sent by p_i at the current round (line 9). Then, p_i consumes messages (line 10), and uses them to modify its local simulation state and compute the message it will send during the next round (line 11).

Lemma 1 *If a task can be solved in $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$, it can be solved in $\mathcal{ARW}[fd : \Omega]$.*

Proof Let us consider a simulated process p_i (p_i executes in $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$). When there is no ambiguity, we use the same identifier p_i, for a simulated process and its simulator.

Let us first observe that no correct simulator p_i can block forever in the loop of lines 6-8. This is an immediate consequence of the eventual leadership property of Ω (the eventually elected process p_ℓ cannot block forever), and the fact that it writes its messages in $MEM[\ell][r][1..n]$.

Let us show that the tournament property TOUR is satisfied at every round. Let us consider two processes that terminate a round $r \geq 0$. It follows from lines 9-10 that (1) p_i has written a message into $MEM[i][r][j]$ and then read $MEM[j][r][i]$, while (2) p_j has written a message in $MEM[j][r][i]$ and then read $MEM[i][r][j]$. As registers are atomic, it follows that either p_i has written into $MEM[i][r][j]$ before p_j has written into $MEM[j][r][i]$, or the opposite. Whatever the

initialization:
(1) $r_i \leftarrow 0$;
(2) $ls\_state_i \leftarrow$ initial state of the local simulated algorithm;
(3) $msgs\_to\_send_i[1..n] \leftarrow$ initial messages to send to each process;
(4) $\forall r > 0 : MEM[i][r][1..n]$ init to $[\perp, ..., \perp]$.

repeat forever
(5) $r_i \leftarrow r_i + 1$;
(6) **repeat** $leader\_val_i \leftarrow MEM[leader_i][r_i][i]$
(7) **until** $(leader\_val_i \neq \perp) \vee (leader_i = i)$
(8) **end repeat**;
(9) $MEM[i][r_i] \leftarrow msgs\_to\_send_i$;
(10) $rec\_msgs_i[1..n] \leftarrow MEM[1..n][r_i][i]$;
(11) $(msgs\_to\_send_i, ls\_state_i) \leftarrow \mathsf{simulate}(ls\_state_i, rec\_msgs_i)$
end repeat.

Figure 2.3: From $\mathcal{ARW}[fd : \Omega]$ to $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$

case, as each process writes before reading, at least one of them reads the message from the other, and consequently $\mathcal{G}^r$ contains (i, j) or (j, i). Let us now consider a process p_j whose simulator crashes during the execution. From the point of view of any process p_i whose simulator is correct, everything appears as if, after the simulator of p_j has crashed, the simulated adversary removes all the messages sent by p_j to p_i and keeps the messages sent by p_i to p_j. Hence, if p_j crashes after round r, we have $(i, j) \in \mathcal{G}^{r'}$ at any round $r' > r$.

Finally, let us consider a time after which all the correct simulators have forever the same correct leader p_ℓ and no more simulator crashes. It follows from lines 6-8, that there is a round r such that, at any round $r' \geq r$, any correct process p_i receives the message sent by p_ℓ. Moreover, crashed processes receive implicitly all messages sent by p_ℓ. It follows that we have $(\ell, i) \in \mathcal{G}^{r'}$ which establishes the SOURCE property of the adversary.

It follows from the previous arguments that, if the task can be solved in the model $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$, it can be solved in $\mathcal{ARW}[fd : \Omega]$. A process with a correct simulator behaves the same way in both models, and a process with a faulty simulator either computes a correct output value or crashes before it has computed an output value (in this case, its entry in the output vector contains $\perp$). $\square_{Lemma\ 1}$

2.7.3 From $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$ to $\mathcal{ARW}[fd : \Omega]$

This section presents a simulation of $\mathcal{ARW}[fd : \Omega]$ on top of $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$ such that, any task that can be solved in $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$ can be solved in $\mathcal{ARW}[fd : \Omega]$. This simulation has the same structure as the simulation of $\mathcal{ARW}$ on top of $\mathcal{SMP}$ described in [2]. Basically, it adds to it the management of the local variables $missed_i$ (defined below) from which Ω is extracted.

Global and local variables of the simulation The shared memory of $\mathcal{ARW}[fd : \Omega]$ is made up of an array of single-writer/multi-reader atomic registers $MEM[1..n]$ such that only p_i can write $MEM[i]$. The simulation associates a sequence number with each read or write operation of a simulated process p_i. To simplify notations, a read of $MEM[\ell]$ by p_i is denoted $\mathsf{read}_i(\ell)$ and a write of v into $MEM[i]$ is denoted $\mathsf{write}_i(v)$.

As in the previous simulation, the procedure $\mathsf{simulate}()$ is used to locally simulate the behavior of p_i from its current step until its next invocation of a communication operation (i.e., a read or a write of the simulated shared memory). The simulation stops just before this invocation. It takes

as input parameters the current local state of p_i (ls_state_i) and the last value read from the shared memory by p_i. This value, saved in $read\_value_i$ (and initialized to $\perp$), is meaningless if the operation is a write. The local variable $next\_op_i$ contains p_i's next read or write operation to be simulated.

The local variable $view_i$ contains all the read/write operations issued by the processes and known by p_i. Such an operation is represented by a triple $(j, seq\_nb, next\_op)$. The simulation algorithm is a full information algorithm and consequently the set $view_i$ increases forever.

The local variable $informed_i$ contains the set of processes which, to the knowledge of p_i, know the last read/write operation it is currently simulating. Finally, the set $missed_i$ (from which Ω is built) contains pairs (k, r) whose meaning is the following: $((k, r) \in missed_i) \Rightarrow$ there is at least one process that, during round r of the simulation, has not received and delivered the message sent by (the simulator of) p_k during that round.

The simulation algorithm The simulation algorithm is described in Figure 2.4. When it starts a new round, the simulator of p_i sends its control local state, i.e., the triple $(i, view_i, missed_i)$ to each other process (line 5). Then (lines 6-10), it considers all the messages it has received during the current round r, and updates accordingly rec_msg_i and $missed_i$.

Lines 11-12 locally implement Ω (see below). The variable $informed_i$ is then updated to take into account what has been learned from the messages just received. Let us notice (line 13) that it follows from TOUR that $(j \notin rec\_from_i) \Rightarrow p_j$ has received p_i's round r message.

initialization:
(1) $ls\_state_i \leftarrow$ initial state of the local simulated algorithm; $read\_value_i \leftarrow \perp$;
(2) $(next\_op_i, ls\_state_i) \leftarrow \mathsf{simulate}(local\_sim\_state_i, read\_value_i)$;
(3) $seq\_nb_i \leftarrow 1$; $informed_i \leftarrow \{i\}$; $missed_i \leftarrow \emptyset$;
(4) $view_i \leftarrow \{(i, seq\_nb_i, next\_op_i)\}$.

round $r = 1, 2, \cdots$ **do**:
(5) $\mathsf{send}(i, view_i, missed_i)$ to each other process;
(6) $rec\_msgs_i \leftarrow$ set of triples $(j, view_j, missed_j)$ received during this round;
(7) $view_i \leftarrow view_i \cup \Big(\bigcup_{(j,view_j,missed_j)\in rec\_msgs_i} view_j\Big)$;
(8) $missed_i \leftarrow missed_i \cup \Big(\bigcup_{(j,view_j,missed_j)\in rec\_msgs_i} missed_j\Big)$;
(9) $rec\_from_i \leftarrow \big\{j \in \{1, \ldots, n\} \ : \ \exists (j, view_j, missed_j) \in rec\_msgs_i\big\} \cup \{i\}$;
(10) $missed_i \leftarrow missed_i \cup \big\{(k, r) \ : \ k \in \{1, \ldots, n\} \setminus rec\_from_i\big\}$;
(11) $min\_missed_i \leftarrow \mathsf{min}\big\{|\{r \ : \ (j, r) \in missed_i\}|, j \in \{1, \ldots, n\}\big\}$;
(12) $\ell d_i \leftarrow \mathsf{min}\big\{j \ : \ |\{r \ : \ (j, r) \in missed_i\}| = min\_missed_i\big\}$;
(13) $informed_i \leftarrow informed_i \cup (\{1, \ldots, n\} \setminus rec\_from_i)$
(14) $\qquad\qquad\qquad\qquad \cup \big\{j \in rec\_from_i \ : \ (i, seq\_nq_i, next\_op_i) \in view_j\big\}$;
(15) **if** $(informed_i = \{1, \ldots, n\})$ **then**
(16) $\quad (next\_op_i, ls\_state_i) \leftarrow \mathsf{simulate}(ls\_state_i, read\_value_i)$;
(17) $\quad$ **if** $(next\_op_i = \mathsf{read}_i(\ell))$ **then**
(18) $\quad\quad$ **if** $(\nexists (\ell, -, \mathsf{write}_\ell(-)) \in view_i)$
(19) $\quad\quad\quad$ **then** $read\_value_i \leftarrow \perp$
(20) $\quad\quad\quad$ **else** $max\_sn\ell_i \leftarrow \mathsf{max}\{sn_\ell, (\ell, sn_\ell, \mathsf{write}_\ell(-)) \in view_i\}$;
(21) $\quad\quad\quad\quad\ read\_value_i \leftarrow v_\ell \ : \ (\ell, max\_sn\ell_i, \mathsf{write}_\ell(v_\ell)) \in view_i$
(22) $\quad\quad$ **end if**
(23) $\quad$ **end if**;
(24) $\quad seq\_nb_i \leftarrow seq\_nb_i + 1$; $informed_i \leftarrow \{i\}$;
(25) $\quad view_i \leftarrow view_i \cup \{(i, seq\_nb_i, next\_op_i)\}$
(26) **end if**.

when $leader_i$ **is read**: return (ℓd_i).

Figure 2.4: Simulation of $\mathcal{ARW}[fd : \Omega]$ in $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$

Then (the simulator of) p_i executes rounds in $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$ until it learns that (the simulators of) all the processes know its last read/write operation (line 15). Then, (line 16) it invokes simulate$(ls_state_i, read_value_i)$. If its (simulated) shared memory operation is a read, the value $read\_value_i$ is the value obtained by this read operation. Otherwise (the simulated operation is a write of p_i), $read\_value_i$ is useless. As already indicated, the invocation of simulate$(ls_state_i, read_value_i)$ simulates then the behavior of p_i until its next read/write operation.

If the operation at which the local simulation stopped is a read of $MEM[\ell]$ (line 17), the local simulator computes, and deposits in $read\_value_i$, the value that will be associated with this read (line 18-22). If p_ℓ has not issued a write, $read\_value_i$ is set to the default value $\perp$ (line 19). Otherwise, $read\_value_i$ is set to the last value written by p_ℓ (line 18-21). Then, whatever the next operation (read or write) of p_i, the local simulator associates a sequence number with it and adds the triple $(i, seq\_nb_i, next\_op_i)$ to $view_i$ (line 24-25). Moreover, as its scope is the simulation of $next\_op_i$, the set $informed_i$ is reset to $\{i\}$.

As previously indicated, the current value (kept in ℓd_i) of the read-only variable $leader_i$, which locally implements Ω, is computed from the set $missed_i$ at lines 11-12. The simulator of p_i (1) computes, for each p_j, the set of rounds at which at least one simulator has not received the round r message sent by p_j's simulator (these are messages suppressed by the adversary); then (2) it associates with each p_j the cardinality of the previous set; and finally, (3) it considers the process p_ℓ for which the adversary has suppressed the least messages (if there are several such processes, ties are solved by using the total order on process identities).

Lemma 2 *If a task can be solved in $\mathcal{ARW}[fd : \Omega]$, it can be solved in $\mathcal{SMP}[adv :$ SOURCE, TOUR$]$.*

Proof The proof consists of four parts: (1) the simulation is non-blocking; (2) the definition of which are the correct/faulty processes in $\mathcal{ARW}[fd : \Omega]$; (3) the definition of the linearization of the read and write operations; and (4) the fact that local variables $leader_i$ implement Ω.

Part 1: the simulation is non-blocking.
Given that the simulation algorithm is a full information algorithm, the *king in two tournaments* Theorem [3][4] states that at least one read/write operation issued by a simulated process p_i is known by all simulators in two simulation rounds (i.e., there is a simulator p_i such that, for any j, we have $next\_op_i \in view_j$ in at most two rounds).

Let us consider three consecutive simulation rounds r, $r+1$ and $r+2$, and p_i a simulator whose message $(i, view_i, missed_i)$ sent at line 5 has reached (directly or indirectly) all the process simulators by the end of round $r+1$. As (due to TOUR), $\mathcal{G}^{r+2}$ contains a tournament, we have one of the following for each $j \neq i$: (a) $j \stackrel{r+2}{\longrightarrow} i$ and in that case, p_i receives its own triple $(i, seq\_nb_i, next\_op_i)$ from p_j and consequently it knows that p_j knows its triple $(i, seq\_nq_i, next\_op_i)$ (line 14); or (b) $\neg(j \stackrel{r+2}{\longrightarrow} i)$ and in that case, we necessarily have $i \stackrel{r+2}{\longrightarrow} j$ and consequently p_i knows that p_j knows its triple. It follows that, at the end of the simulation round $r+2$, p_i terminates the simulation of its read or write operation $next\_op_i$ (line 15-line 23). As there is a bounded number of processes, there is at least one process that eventually executes until it computes its local result. It follows that the simulation is non-blocking.

[4] This theorem extends a theorem on graph tournament by Landau [55] to the case where consecutive tournaments can be different.

Part 2: correct and faulty processes in $\mathcal{ARW}[fd : \Omega]$.
As each message graph $\mathcal{G}^r$ contains a tournament, it follows that the relation $\overset{\infty}{\rightsquigarrow}$ (introduced in Section 2.6.1) defines a total order on the equivalence classes of the relation $\overset{\infty}{\leftrightsquigarrow}$. Hence, there is a single set X of strongly correct process simulators (i.e., X has no input edge in $SC(G)$). This set X contains exactly all the process simulators whose messages sent at line 5 are always eventually received (at some round, directly or indirectly) by all other process simulators.

It follows from the previous reasoning (Part 1) on the fact that the simulation is non-blocking, and the condition $informed_i = \{1, ..., n\}$ (line 15), that each process in X simulates any number of its operations. Hence, those processes are correct in $\mathcal{ARW}[fd : \Omega]$. Differently, for each process p_j such that $j \in \{1, ..., n\} \setminus X$, there is a round after which the predicate $informed_j = \{1, ..., n\}$ is never satisfied, and consequently the simulation of the process p_j stops progressing. Hence, each weakly correct simulator p_j such that $j \in \{1, ..., n\} \setminus X$ simulates a faulty process in $\mathcal{ARW}[fd : \Omega]$.

Part 3: definition of the linearization of the read and write operations.
A write operation is linearized at the first of the two following time instants: (1) τ_1 the end of the simulation of this write operation (i.e., when the condition line 15 is satisfied by the associated simulator), and (2) τ_2 the time instant just before the linearization point of the simulation of the first read operation returning the written value. If none of these two instants ever happen, then the write is never linearized and the corresponding simulated process appears as crashed.

The linearization of a read operation is close to but different from the one of a write operation. A read operation is linearized at the first of these two time instants: (1) τ_1 the end of this read operation simulation (when the condition line 15 is satisfied by the associated simulator), and (2) τ_2 the time instant just before the linearization point of the simulation of the first write operation which overwrites the read value. However, if the instant τ_1 never happens, then the read is never linearized (even if τ_2 exists) and the simulated process appears as crashed. (Remark that, thanks to the fact that a simulator selects the freshest value it knows to prepare the value returned by a simulated read operation, and to the fact that, at the instant of the linearization of a write (or read) operation, all processes have the corresponding written (or read) value in their views, the read value cannot have been effectively overwritten before the beginning of the read operation returning it.)

As the reading (resp. overwriting) of a written (read) value cannot occur neither before the start of the corresponding write (read) operation, the linearization point of this operation occurs after the start of the operation. Moreover, the selection of the first among τ_1 and τ_2 for both types of operations implies that the linearization point occurs at the latest at the end of its simulation, and that (a) a read is always linearized after the writing of the value it returns, (b) the overwriting of a value is always linearized after all read operations that return it.

Part 4: the local variables $leader_i$ implement Ω.
It follows from the property SOURCE that there is a process p_s and a round r_0 such that, from r_0, no message from p_s is removed by the adversary. In particular, after some round $r_s \geq r_0$, all the messages sent by (the simulator of) p_s are received by all the strongly correct processes. Let $S \subseteq X$ be the set of processes p_s satisfying this property (eventually all their messages are –directly– received by all the strongly correct processes). Let $r_S = \max\{r_s : s \in S\}$. (Let us notice that an arbitrary number of messages from the processes in S to processes which are not strongly correct can be suppressed by the adversary.)

As, at any round $r \geq r_S$, no message from (the simulator of) a process in S to (the simulator of) a strongly correct process is suppressed, no simulator of a strongly correct process p_i adds a pair (s, r), $s \in S$, to its set $missed_i$ (line 10).

Let us observe that: (a) the simulator of any process p_i adds all the set $missed$ it receives to its own set $missed_i$ (line 8); (b) due to the definition of "strongly correct" simulator, messages are eventually propagated (directly or indirectly) in each direction between each pair of strongly correct simulators; and (c) after some finite time, no strongly correct simulator receive message from a weakly correct simulator.

It follows from the previous observations that, for each $s \in S$, the values $|\{r : (r,s) \in missed_i\}|$ eventually stabilize at the same finite value at each strongly correct simulator p_i, while, for each $j \in \{1,...,n\} \setminus S$, the value $|r : (r,j) \in missed_i\}|$ never stops increasing. Hence, the same eventual leader is elected at all strongly correct processes by the code of lines 11-12, and consequently the correct simulated processes inherits the same eventual leader. Moreover, as the process simulators in S are strongly correct, the elected leader is a process which is correct in $\mathcal{ARW}[fd : \Omega]$. $\square_{Lemma\ 2}$

2.7.4 SOURCE + TOUR is a Characterization of Ω in $\mathcal{ARW}$

Theorem 5 *A task can be solved in $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$ iff it can be solved in $\mathcal{ARW}[fd : \Omega]$.*

Proof The proof follows immediately from Lemma 1 and Lemma 2. $\square_{Theorem\ 5}$

Remark Let us remark that it is not possible to conclude from the previous theorem and the fact that Ω is the weakest failure detector to solve consensus in $\mathcal{ARW}$, that the property SOURCE + TOUR defines a weakest message adversary AD allowing consensus to be solved in $\mathcal{SMP}[adv : \text{AD}]$. It remains possible that a property AD weaker than SOURCE + TOUR allows consensus to be solved in $\mathcal{SMP}[adv : \text{AD}]$. Said differently nothing allows us to claim that the "granularity on the properties which can be defined to constrain message adversaries" is the same as the "granularity on the information on failures" provided by failure detectors.

Let CONS be the minimal message adversary property that allows consensus to be solved in $\mathcal{SMP}[adv : \text{CONS}]$. As consensus is solvable in $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$, it follows that $\mathcal{SMP}[adv : \text{SOURCE, TOUR}]$ is at least as powerful as $\mathcal{SMP}[adv : \text{CONS}]$. On another side, as (1) a read/write register can be implemented from consensus, and (2) consensus cannot be solved in $\mathcal{SMP}[adv : \text{TOUR}]$, it follows that $\mathcal{SMP}[adv : \text{CONS}]$ is strictly more powerful than $\mathcal{SMP}[adv : \text{TOUR}]$.

2.8 SOURCE is a Characterization of Ω in $\mathcal{AMP}$

2.8.1 From $\mathcal{AMP}[fd : \Omega]$ to $\mathcal{SMP}[adv : \text{SOURCE}]$

The algorithm described in Figure 2.5 presents a simulation (for tasks) of $\mathcal{SMP}[adv : \text{SOURCE}]$ on top of $\mathcal{AMP}[fd : \Omega]$. Its principles are close to the ones of the simulation of Figure 2.3. The algorithm ensures that the eventual leader p_ℓ satisfies the property SOURCE. Hence, there are strongly correct processes and the eventual leader is one of them. The aim of the simulation algorithm is then to eventually withdraw all the messages except the ones from the leader.

Local variables of the simulation As in the previous simulations, r_i is the locally simulated round number; $msgs\_to\_send_i[j]$ (initialized to $\perp$) contains the next simulated message to be sent to p_j; $rec\_msg_i[r]$ contains the simulated messages received at round r; $sim\_rec\_msgs_i[x]$ contains the message received from the process p_x currently considered as the leader by p_i; $leader_i$ is the read-only variable provided by Ω.

```
(1)  r_i ← 0; sim_rec_msgs_i[1,...,n] ← [⊥,...,⊥];
(2)  (msgs_to_send_i[1,...,n], ls_state_i) ← simulate(sim_rec_msgs_i);
(3)  for each r > 0 do rec_msgs_i[r][1,...,n] ← [⊥,...,⊥] end for;
(4)  repeat forever
(5)    r_i ← r_i + 1;
(6)    for each j ∈ {1,...,n} do send(r_i, msgs_to_send_i[j]) to p_j end for;
(7)    repeat cur_ℓd_i ← leader_i
(8)      until (cur_ℓd_i = i ∨ rec_msgs_i[r_i][cur_ℓd_i] ≠ ⊥)
(9)    end repeat;
(10)   sim_rec_msgs_i[cur_ℓd_i] ← rec_msgs_i[r_i][cur_ℓd_i];
(11)   (msgs_to_send_i[1,...,n], ls_state_i) ← simulate(sim_rec_msgs_i);
(12)   sim_rec_msgs_i[1,...,n] ← [⊥,...,⊥]
(13) end repeat.

when (r, m) received from p_j:  rec_msgs_i[r][j] ← m.
```

Figure 2.5: From $\mathcal{AMP}[fd : \Omega]$ to $\mathcal{SMP}[adv : \text{SOURCE}]$

The simulation algorithm The procedure simulate() takes as input parameter the simulated messages received by p_i at the current round, and simulates the local algorithm until the next sending of messages by p_i. This procedure returns the simulated messages to be sent at the beginning of the next round.

After the initialization stage (lines 1-3), the local simulator of p_i enters a loop whose each body execution simulates a round of the synchronous system. It first sends the messages that p_i has to send at the current round (line 6). Then it waits until it has received a message from its current leader or it is its own leader (lines 7-9). When this occurs, it retrieves the message sent by its current leader (line 10) and invokes the procedure simulate() with this message as input parameter, before proceeding to the simulation of the next synchronous round.

Lemma 3 *If a task can be solved in* $\mathcal{SMP}[adv : \text{SOURCE}]$*, it can be solved in* $\mathcal{AMP}[fd : \Omega]$.

Proof Let us first show that no correct process remains blocked forever in the loop lines of 7-9. Indeed, there is a finite time τ after which an eventual leader (say p_ℓ) is elected by Ω at each process. It then follows from the first part of the predicate of line 8 that p_ℓ cannot remain blocked at line 8, and consequently executes rounds forever. Moreover, as its messages are eventually received at each round by all correct processes, it follows that there is a time after which the second part of the predicate of line 8 is always satisfied by these processes. Consequently, none of them can remain blocked forever at line 8.

The previous reasoning shows also that the eventual leader elected by Ω behaves as a source, consequently the property SOURCE is satisfied in the simulated synchronous system. $\square_{Lemma\ 3}$

2.8.2 From $\mathcal{SMP}[adv : \text{SOURCE}]$ to $\mathcal{AMP}[fd : \Omega]$

The simulation algorithm is described in Figure 2.6. It is similar to the algorithm of Figure 2.4 (which simulates $\mathcal{ARW}[fd : \Omega]$ on top of $\mathcal{SMP}[adv : \text{SOURCE}, \text{TOUR}]$).

Local variables of the simulation The local variables ls_state_i, $view_i$, rec_from_i, and $missed_i$ have the same meaning as in Figure 2.4. The local variable $msgs\_to\_rec_i$ contains messages to be consumed by the simulated process (it corresponds to $read\_value_i$ in Figure 2.4). The variable $msgs\_to\_send_i$ contains the messages to be sent in the next simulation round (it

corresponds to $next\_op_i$ in Figure 2.4). The variable $msgs\_received_i$ is a new variable containing the messages already received by the simulated process p_i. Finally, ℓd_i is the local variable containing the current local value of Ω built by the algorithm.

The simulation algorithm As in the simulation of Figure 2.4, lines 1-4 are an initialization stage. Similarly to previous simulations, the procedure simulate() locally simulates the process p_i. It takes messages to be consumed by p_i as input parameter and returns the next set of messages to be sent.

The simulation algorithm is a full information algorithm. During each simulation round r, the simulator of p_i first sends its control local state to each other process, and waits for the same information from them (lines 5-6). Then, according to the messages it has received during the current round, it updates $view_i$, $missed_i$, and rec_from_i (lines 7-10). As in Figure 2.4, it also computes the identity ℓd_i of its current candidate to be the eventual leader (lines 11-12).

If a simulation message has been received from the process $p_{\ell d_i}$, the simulator of p_i strives to make p_i progress. It considers the last message sent by $p_{\ell d_i}$ to p_i (triple $(\ell d_i, i, m)$), and adds it to the set $msgs\_to\_rec_i$ (lines 14-15). Then, if the messages p_i has to send are known by its current leader $p_{\ell d_i}$ (line 16), the procedure simulate() is invoked to make p_i progress (line 17), and the local control variables $msgs\_received_i$ and $view_i$ are updated accordingly (line 18).

initialization:
(1) $ls\_state_i \leftarrow$ initial state of the local simulated algorithm;
(2) $msgs\_to\_rec_i \leftarrow \emptyset$; $msgs\_received_i \leftarrow \emptyset$;
(3) $(msgs\_to\_send_i, ls\_state_i) \leftarrow \mathsf{simulate}(ls\_state_i, msgs\_to\_rec_i)$;
(4) $view_i \leftarrow msgs\_to\_send_i$; $missed_i \leftarrow \emptyset$; $\ell d_i \leftarrow i$.

round $r = 1, 2, \cdots$ **do**:
(5) $\mathsf{send}(i, view_i, missed_i)$ to each other process;
(6) $rec\_msgs_i \leftarrow$ set of triples $(j, view_j, missed_j)$ received during this round;
(7) $view_i \leftarrow view_i \cup \Big(\bigcup_{(j,view_j,missed_j)\in rec\_msgs_i} view_j\Big)$;
(8) $missed_i \leftarrow missed_i \cup \Big(\bigcup_{(j,view_j,missed_j)\in rec\_msgs_i} missed_j\Big)$;
(9) $rec\_from_i \leftarrow \big\{j \in \{1,\ldots,n\} \ : \ \exists (j, view_j, missed_j) \in rec\_msgs_i\big\} \cup \{i\}$;
(10) $missed_i \leftarrow missed_i \cup \big\{(k,r) \ : \ k \in \{1,\ldots,n\} \setminus rec\_from_i\big\}$;
(11) $min\_missed_i \leftarrow \mathsf{min}\big\{|\{r \ : \ (j,r) \in missed_i\}|, j \in \{1,\ldots,n\}\big\}$;
(12) $\ell d_i \leftarrow \mathsf{min}\big\{j \ : \ |\{r \ : \ (j,r) \in missed_i\}| = min\_missed_i\big\}$;
(13) **if** $(\ell d_i \in rec\_from_i)$ **then**
(14) **let** $view_{\ell d_i}$ **be such that** $(\ell d_i, view_{\ell d_i}, missed_{\ell d_i}) \in rec\_msgs_i$;
(15) $msgs\_to\_rec_i \leftarrow msgs\_to\_rec_i \cup \{(\ell d_i, i, m) \ : \ (\ell d_i, i, m) \in view_{\ell d_i}\}$;
(16) **if** $(msgs\_to\_send_i \subseteq view_{\ell d_i})$ **then**
(17) $(msgs\_to\_send_i, ls\_state_i) \leftarrow \mathsf{simulate}(ls\_state_i, msgs\_to\_rec_i \setminus msgs\_received_i)$;
(18) $msgs\_received_i \leftarrow msgs\_to\_rec_i$; $view_i \leftarrow view_i \cup msgs\_to\_send_i$
(19) **end if**
(20) **end if**.

when $leader_i$ **is read**: return (ℓd_i).

Figure 2.6: Simulation of $\mathcal{AMP}[fd : \Omega]$ in $\mathcal{SMP}[adv : \text{SOURCE}]$

Lemma 4 *If a task can be solved in $\mathcal{AMP}[fd : \Omega]$, it can be solved in $\mathcal{SMP}[adv : \text{SOURCE}]$.*

Proof Preliminary definition on simulators in $\mathcal{SMP}[adv : \text{SOURCE}]$.
Let S be the set of processes which satisfy the property SOURCE. As, by assumption there is at least one source, we have $S \neq \emptyset$. Moreover, due to the definition of the set $\mathcal{SC}$ of strongly correct

simulators we have $S \subseteq \mathcal{SC}$. Let S' be the set of processes which, albeit they are not necessarily source, appear as sources to all processes of $\mathcal{SC}$. Hence we have $S \subseteq S' \subseteq \mathcal{SC}$, and $S' \neq \emptyset$.

The variables $leader_i$ implement Ω.
According to the definition of $\mathcal{SC}$, there is a round r_0 after which no more messages from weakly correct simulators are received (directly or indirectly) by a strongly correct simulator. Let $r_1 = \max\{r_s, s \in S'\}$ where r_s is the first round after which no message sent by p_s to a strongly correct simulator is eliminated. As, after r_1, each strongly correct simulator receives at every round a message from each simulator in S', it follows that none of them adds a pair $(s, r), r \geq r_1, s \in S'$ in its variable $missed_i$ at line 10. After $r_2 = \max\{r_0, r_1\}$, the only pairs $(s, r), s \in S'$ $(r < r_1)$ that are added by a strongly correct simulator in its variable $missed_i$ are those that have been added by other strongly correct simulators at line 8 or line 11 before r_2. Since strongly correct simulators are infinitely often able to transmit (directly or not) messages to each other, there is a round $r_3 \geq r_2$ such that any strongly correct simulator p_i has received (directly or not) during a round $r_j \geq r_2$ the information contained in the variable $missed_j$ from each other strongly correct simulator p_j. After r_3, for any $s \in S'$, the number of pairs (s, r) in the variables $missed_i$ of all strongly correct simulators p_i is the same and does not increase anymore.

For each simulator p_i, $i \notin S'$, there is an infinite number of rounds r such that p_i's message is not received during round r by at least one of the strongly correct simulator p_j, and accordingly, this simulator adds a pair (i, r) to its variable $missed_j$ during round r at line 10. As the strongly correct simulators communicate (directly or not) infinitely often with each other, all of them eventually add this pair to their variable $missed$ during r (at line 10) or later (at line 8). Consequently, for each such simulator p_i, $i \notin S'$, the number of pairs (i, r) in the variable $missed_j$ of every strongly simulator p_j increases forever.

It follows from the previous discussion that the minimal number of rounds missed by a simulator (as calculated at line 12, and using simulator identity to do tie-breaking) eventually becomes and remains the same at each strongly correct simulator. Let ℓd denote this simulator identity. As it is the identity that is eventually always returned when $leader_i$ is read by any simulated process p_i whose simulator is strongly correct, the eventual unicity property of Ω is ensured for these processes. The next paragraph shows that the set of strongly correct simulators corresponds exactly to the set of correct simulated processes. As $p_{\ell d}$ is strongly correct, the elected process is a correct process, which concludes the proof of Ω.

Correct and faulty (simulated) processes in $\mathcal{AMP}[fd : \Omega]$.
It follows from the previous paragraphs that each strongly correct simulator p_i is always eventually able to transmit (directly or not) a new message m to $p_{\ell d}$, and then eventually receive (directly) a message from $p_{\ell d}$ containing m. Hence the conditions of line 13 and line 16 are fulfilled an infinite number of times and, consequently, the corresponding simulator can always issue enough steps (line 17) to progress in the simulated code.

Hence, the correct simulated processes and the faulty simulated processes are the ones simulated by the strongly correct and weakly correct simulators, respectively.

Linearization of communication operations.
Let us consider a simulated process p_i that sends a message m to a simulated process p_j. This operation is disseminated to each simulator by p_i's simulator at line 5. Then a simulator considers this simulated message m only at line 17 when the second input parameter of its invocation of simulate() contains the message m. (Let us observe, that this message m arrives at a simulator p_k from its current leader ℓd_k, lines 13 and 16.)

Let τ_1 be the time of the first invocation of simulate() by a simulator such that m belongs to

the second input parameter of this invocation, where $\tau_1 = \infty$ if there is no such invocation. Let τ_2 be the time at which the simulator of p_i starts the execution of simulate() (line 17) after it has disseminated m, where $\tau_2 = \infty$ if there is no such invocation.

The sending of m is linearized at time $\min(\tau_1, \tau_2)$ (let us notice that the simulation of p_i does not progress between the sending of m by p_i and its linearization point). If $\min(\tau_1, \tau_2) = \infty$, the send of m is linearized after the receiver p_j has computed its result.

The reception of m is linearized at the time of the invocation by p_j of simulate() whose second input parameter contains the message m, or after p_j has computed its result if there is no such invocation. $\square_{Lemma\ 4}$

2.8.3 SOURCE is a characterization of Ω in $\mathcal{AMP}$

Theorem 6 *A task can be solved in $\mathcal{AMP}[fd : \Omega]$ iff it can be solved in $\mathcal{SMP}[adv :$ SOURCE, FAIR$]$.*

Proof The proof follows directly from Lemma 3 and Lemma 4. $\square_{Theorem\ 6}$

2.9 QUORUM is a Characterization of Σ in $\mathcal{AMP}$

This section shows that the computing models $\mathcal{SMP}[adv :$ QUORUM$]$ and $\mathcal{AMP}[fd : \Sigma]$ have the same computational power for tasks.

2.9.1 The Property QUORUM

Let us remember that $\mathcal{SC}$ is the set of strongly correct processes in the considered synchronous message-passing system (processes of which an infinite number of messages are received by each other process). The property QUORUM is defined as follows:

$$\big[\forall\, i, j : \forall\, r_i, r_j : (\{k : k \xrightarrow{r_i} i\} \cap \{k : k \xrightarrow{r_j} j\} \neq \emptyset)\big] \ \wedge\ (\mathcal{SC} \neq \emptyset).$$

This property is a statement of Σ suited to the context of round-based synchronous message-passing systems prone to message adversaries. Given any pair of processes p_i and p_j, its first part states that, whatever the synchronous rounds r_i and r_j executed by p_i and p_j, respectively, there is a process p_k whose messages to p_i at round r_i and to p_j at round r_j are not eliminated by the adversary (intersection property). The second part states that there is at least one process whose messages are infinitely often received by each other process (liveness property). Theorem 7 will show that this formulation of Σ is correct for the equivalence of $\mathcal{AMP}[fd : \Sigma]$ and $\mathcal{SMP}[adv :$ QUORUM$]$ for task solvability.

2.9.2 From $\mathcal{AMP}[fd : \Sigma]$ to $\mathcal{SMP}[adv :$ QUORUM$]$

The simulation algorithm is described in Figure 2.7. It has the same local variables as, and is very close to, the one of Figure 2.5. In addition to the local output of the failure detector Σ, which is denoted qr_i, the only modifications are the lines 7-10 which differ in both algorithms.

The simulator of p_i waits until it has received a message from each process that appears in its current quorum qr_i (lines 7-9). It then invokes the procedure simulate() with these messages as input (line 10).

The principle of this simulation is the following: after some time, the simulated message adversary suppresses all the messages sent by processes that do not belong to a quorum, but is prevented from suppressing the messages sent by processes belonging to quorums.

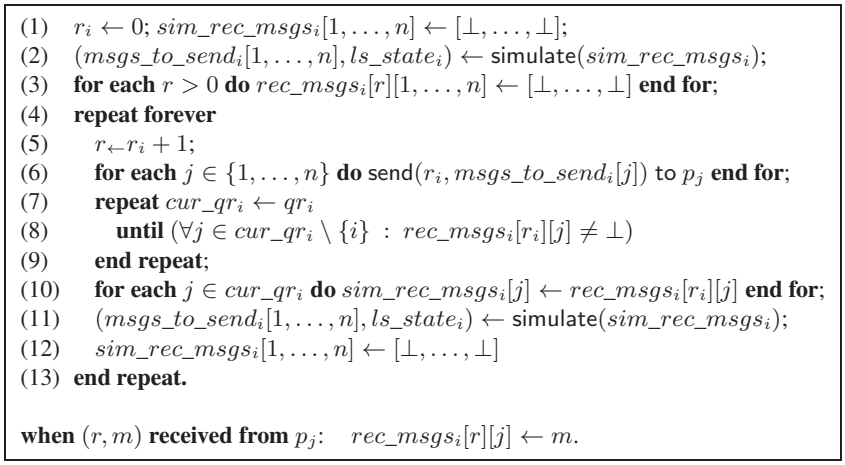


Figure 2.7: From $\mathcal{AMP}[fd : \Sigma]$ to $\mathcal{SMP}[adv : \text{QUORUM}]$

Lemma 5 *If a task can be solved in $\mathcal{SMP}[adv : \text{QUORUM}]$, it can be solved in $\mathcal{AMP}[fd : \Sigma]$.*

Proof The proof that no simulator of a process p_i remains forever blocked in a round r_i follows directly from the fact that (1) each process simulator sends a message to each other process simulator at every round (line 6), and (2) each quorum qr_i eventually contains only correct simulators (liveness of Σ).

Let qr_i^r be the value of qr_i that allows p_i to exit the repeat loop during the simulation of round r (lines 7-9). It follows from line 8 and line 10 that $qr_i^r = \{k : k \xrightarrow{r_i} i\}$. Moreover, it follows from the intersection property provided by Σ that $\forall\, i,j,r_i,r_j : qr_i^{r_i} \cap qr_j^{r_j} \neq \emptyset$. The first part of the property QUORUM, namely, $\forall\, i,j,r_i,r_j : (\{k : k \xrightarrow{r_i} i\} \cap \{k : k \xrightarrow{r_j} j\} \neq \emptyset)$, is consequently satisfied.

Let us now show that $\mathcal{SC} \neq \emptyset$. To that end, let us first observe that it follows from the intersection property of Σ that $\forall\, i,j,\forall\, r, \exists k(i,j,r)$ such that $k(i,j,r) \xrightarrow{r} i \ \wedge\ k(i,j,r) \xrightarrow{r} j$. As $\{k(i,j,r)\}_{r>0} \subseteq \{1,...,n\}$, it follows that there is some $k'(i,j)$ which appears infinitely often in the sequence $k(i,j,1),\ k(i,j,2)$, ... Hence, we have $k'(i,j) \overset{\infty}{\rightsquigarrow} i \wedge k'(i,j) \overset{\infty}{\rightsquigarrow} j$. As this is true for any pair (i,j), it follows that the graph G, whose set of vertices is $\{1,...,n\}$ and edges are defined by the relation $\overset{\infty}{\rightsquigarrow}$, has a single strongly connected component without input edges. As this strongly connected component defines the set of strongly correct processes, this set is not empty, which concludes the proof of the lemma. $\square_{Lemma\ 5}$

2.9.3 From $\mathcal{SMP}[adv : \text{QUORUM}]$ to $\mathcal{AMP}[fd : \Sigma]$

The simulation algorithm is described in Figure 2.8. It is very close to the simulation of $\mathcal{AMP}[fd : \Omega]$ on top of $\mathcal{SMP}[adv : \text{SOURCE}]$ presented in Figure 2.6. It has the same local variables, except the variable $missed_i$ which is now useless. The value returned when qr_i is read by a simulated process p_i is now the current value of the set rec_from_i.

The only other difference appears at lines 9-10. The simulation of the simulated process p_i (invocation of the procedure simulate() at lines 11) is now constrained by the predicate of line 9 which states that the messages that p_i wants to send (the messages saved in $msg\_to\_send_i$) must be known by at all the simulators defining the current quorum of p_i (set rec_from_i). When this is satisfied, the set of messages to be received by p_i in the next invocation of simulate() is

redefined (line 11) to include the last simulated messages sent to p_i by processes p_j such that $j \in rec\_from_i$.

initialization:
(1) $ls\_state_i \leftarrow$ initial state of the local simulated algorithm;
(2) $msgs\_to\_rec_i \leftarrow \emptyset$; $msgs\_received_i \leftarrow \emptyset$;
(3) $(msgs\_to\_send_i, ls\_state_i) \leftarrow \mathsf{simulate}(ls\_state_i, msgs\_to\_rec_i)$;
(4) $view_i \leftarrow msgs\_to\_send_i$; $rec\_from_i \leftarrow \{1, \ldots, n\}$.

round $r = 1, 2, \cdots$ **do**:
(5) $\mathsf{send}(i, view_i)$ to each other process;
(6) $rec\_msgs_i \leftarrow$ set of pairs $(j, view_j)$ received during this round;
(7) $view_i \leftarrow view_i \cup \left(\bigcup_{(j,view_j) \in rec\_msgs_i} view_j\right)$;
(8) $rec\_from_i \leftarrow \{j \in \{1, \ldots, n\} : \exists (j, view_j) \in rec\_msgs_i\} \cup \{i\}$;
(9) **if** $(msgs\_to\_send_i \in \bigcap_{(j,view_j) \in rec\_msgs_i} view_j)$ **then**
(10) $msgs\_to\_rec_i \leftarrow msgs\_to\_rec_i \cup \{(j, i, m) : (j, view_j) \in rec\_msgs_i \wedge (j, i, m) \in view_j\}$;
(11) $(msgs\_to\_send_i, ls\_state_i) \leftarrow \mathsf{simulate}(ls\_state_i, msgs\_to\_rec_i \setminus msgs\_received_i)$;
(12) $msgs\_received_i \leftarrow msgs\_to\_rec_i$; $view_i \leftarrow view_i \cup msgs\_to\_send_i$
(13) **end if**.

when qr_i **is read**: $\mathsf{return}(rec\_from_i)$.

Figure 2.8: Simulation of $\mathcal{AMP}[fd : \Sigma]$ in $\mathcal{SMP}[adv :$ QUORUM$]$

Lemma 6 *If a task can be solved in* $\mathcal{AMP}[fd : \Sigma]$*, it can be solved in* $\mathcal{SMP}[adv :$ FAIR, QUORUM$]$.

Proof Part 1: Correct and faulty simulated processes.
According to the definition of $\mathcal{SC}$ and to the second part of QUORUM property, we have $\mathcal{SC} \neq \emptyset$ and $\forall i \in \mathcal{SC}, \forall j \in \{1, \ldots, n\} : i \overset{\infty}{\rightsquigarrow} j$. Let p_i be a simulated process p_i whose simulator is strongly correct. As its messages are always received by every process, they are received by the processes in its local set rec_from_i. Moreover, as the simulation algorithm is a full information algorithm, it eventually receives from each process in rec_from_i the messages whose it is simulating the sending. The condition of line 9 becomes then satisfied, and the simulator of p_i is allowed to progress in the simulation of p_i (line 11). Hence, no strongly correct simulator can block forever in the simulation of its simulated process p_i.

According to (1) the intersection property of QUORUM, and (2) the fact that $\mathcal{SC} \neq \emptyset$ implies that $\exists r : \forall r' \geq r, \forall i \in \mathcal{SC} : \{k : k \overset{r'}{\rightsquigarrow} i\} \subseteq \mathcal{SC}$, it follows that $\forall r > 0, \forall i \in \{1, \ldots, n\} : \{k : k \overset{r}{\rightsquigarrow} i\} \cap \mathcal{SC} \neq \emptyset$ (A). Moreover, there is a round after which no message from a weakly correct process reaches a strongly correct process (B). Finally, (predicate at line 9) to progress in the simulation, a simulator has to receive from each simulator in its current set rec_from_i a copy of the messages $msgs\_to\_send_i$ whose it is simulating the sending (C). It follows from A, B, and C that, eventually, the predicate at line 9 of any weakly correct process remains false forever (because its set $msgs\_to\_send_i$ is never received by a strongly correct process). Consequently, there is a finite time after which, all weakly correct simulators stop progressing in the simulation (while they forever execute rounds, they never execute lines 10-12).

According to the previous discussion, a correct (resp., faulty) process in $\mathcal{AMP}[fd : \Sigma]$ is a process whose simulator is strongly (resp., weakly) correct.

Part 2: the local variables rec_from_i implement Σ.
The intersection property of Σ comes directly from the first predicate defining the property QUORUM. The liveness property of Σ is a consequence of Item (2) noticed above, and the fact that the

correct processes in the simulated system $\mathcal{AMP}[fd : \Sigma]$ are exactly those whose simulators are strongly correct in $\mathcal{SMP}[adv : \text{QUORUM}]$.

Part 3: The linearization points of the communication operations are defined the same way as in the proof of Lemma 4. $\square_{Lemma\ 6}$

2.9.4 QUORUM is a Characterization of Σ in $\mathcal{AMP}$

Theorem 7 *A task can be solved in $\mathcal{SMP}[adv : \text{QUORUM}]$ if and only if it can be solved in $\mathcal{AMP}[fd : \Sigma]$.*

Proof The proof follows immediately from Lemma 5 and Lemma 6. $\square_{Theorem\ 7}$

2.10 SOURCE + QUORUM Characterizes $\Sigma + \Omega$ in $\mathcal{AMP}$

Let us notice that the properties SOURCE and QUORUM are independent of one another in the sense that none of them can be obtained from the other. It follows that the power provided by SOURCE and the power provided by QUORUM can be added. More specifically, we have the following:

- A merge of the simulation of $\mathcal{SMP}[adv : \text{SOURCE}]$ in $\mathcal{AMP}[fd : \Omega]$ (Figure 2.5) with the simulation of $\mathcal{SMP}[adv : \text{QUORUM}]$ in $\mathcal{AMP}[fd : \Sigma]$ (Figure 2.7) provides a simulation $\mathcal{SMP}[adv : \text{SOURCE, QUORUM}]$ in $\mathcal{AMP}[fd : \Sigma,\ \Omega]$. The difference between this simulation and the one of Figure 2.5 (or Figure 2.7) is at lines 7-10 which become

  (7) **repeat** $cur\_\ell d_i \leftarrow leader_i$; $cur\_qr_i \leftarrow qr_i$
  (8) **until**$[(\forall j \in cur\_qr_i \setminus \{i\} : rec\_msgs_i[r_i][j] \neq \bot)$
  $\wedge\ (cur\_\ell d_i = i \vee rec\_msgs_i[r_i][cur\_\ell d_i] \neq \bot)]$
  (9) **end repeat**;
  (10) **for each** $j \in cur\_qr_i \cup cur\_ld_i$ **do** $sim\_rec\_msgs_i[j] \leftarrow rec\_msgs_i[r_i][j]$ **end for**.

  The proof is the same as in Lemma 5 augmented by the fact that the eventual leader elected by Ω verifies the property SOURCE as shown in Lemma 3.

- Similarly, it is sufficient to add the management of $missed_i$ and the procedure to query Ω (as done at lines 8-11 of Figure 2.6) to the simulation of $\mathcal{AMP}[fd : \Sigma]$ in $\mathcal{SMP}[adv : \text{QUORUM}]$ (Figure 2.8) in order to provide a simulation of $\mathcal{AMP}[fd : \Sigma,\ \Omega]$ in the model $\mathcal{SMP}[adv : \text{SOURCE, QUORUM}]$.

  The linearization points and the proof of the properties of Σ are the same as in Lemma 6, while the proof of the properties of Ω follows the one of Lemma 4. Let us finally notice that it follows directly from the properties SOURCE and QUORUM that a process verifying the SOURCE property appears eventually in all the simulated quorums.

Theorem 8 then follows:

Theorem 8 *A task can be solved in $\mathcal{SMP}[adv : \text{SOURCE, QUORUM}]$ iff it can be solved in $\mathcal{AMP}[fd : \Sigma,\ \Omega]$.*

2.11 Concluding Remarks on How to Relate Failure Detector Enriched Models and Iterated Models

In this chapter we proposed new approaches to define variants of two well known iterated models, namely the iterated immediate snapshot model $\mathcal{IIS}$ in Sections 2.1 to 2.5 and the synchronous message-passing model weakened by message adversaries in Sections 2.6 to 2.10. The presented techniques allow us to obtain models that are equivalent in terms of computability to the classic non-iterated wait-free models enriched with common failure detectors.

The validity of the presented techniques relies on generic simulations that have to be slightly adapted to the considered failure detector. Moreover, in both iterated models, the proposed transformations rely heavily on the notion of strongly correct processes which appears to us as essential to the understanding of computability issues in these models. This notion has been instrumental in our work on $\mathcal{IIS}$ to define failure detectors in a syntactic manner, more generic than what has been proposed in [74] and allowed us to circumvent the impossibilities presented by [73]. More generally, it is the notion of strongly correct processes that allows, in the two considered iterated models, to capture the processes that have the ability of communicating their state to the whole system infinitely often. When simulating a classic non-iterated model in one of these two iterated models, the strongly correct processes are the ones able to simulate correct processes.

Chapter 3

k-Set Agreement and s-Simultaneous Consensus in Message-passing systems

This chapter presents and studies a family of computability problems, named (s, k)-simultaneous set agreements ((s, k)-SSA), that generalizes both the k-set agreements [22] and the s-simultaneous consensus [4]. It is shown in [17] that, while these two problems are equivalent (for $k = s$) in asynchronous shared memory models, in asynchronous message-passing systems, solving the s-simultaneous agreement is strictly harder than solving the k-set agreement in the general case.

In the work presented here and published in [84], we show, using a different technique based on failure detectors, that for a given integer K, the family of (s, k)-SSA problems such that $K = s \times k$ forms a strong hierarchy from a computability point of view. We also extend this result to the more general family of asymmetric simultaneous set-agreement problems.

Section 3.1 introduces the (s, k)-SSA problem and defines the failure detector $Z_{s,k}$ that is sufficient to solve it in $\mathcal{AMP}$. Section 3.2 presents a $Z_{s,k}$-based algorithm that solves the (s, k)-SSA problem in that model, Section 3.3 shows that the quorum part of $Z_{s,k}$ is necessary to solve the (s, k)-SSA problem in $\mathcal{AMP}$ and Section 3.4.4 presents a generalization of (s, k)-SSA and studies the relations, in terms of computability, of this set of problems. Finally Section 3.5 concludes the chapter.

3.1 Computation Model, (s, k)-SSA Problem, and the Failure Detector $Z_{s,k}$

3.1.1 The s-Simultaneous k-Set Agreement –(s, k)-SSA– Problem

The s-simultaneous k-set agreement problem (in short (s, k)-SSA) consists in the simultaneous execution of s instances of the k-set agreement problem. Moreover, each process proposes the same value to each instance of the k-set agreement problem. The (s, k)-SSA problem is defined by the three following properties.

- Termination. Every correct process decides.
- Validity. A decided value is a pair (c, v) where $1 \leq c \leq s$ and v is a value proposed by a process.
- Agreement. For any $c \in \{1, ..., s\}$, there are at most k different values v such that (c, v) is decided.

It is easy to see that at most $K = sk$ different values v are decided, and consequently, any algorithm solving the (s, k)-SSA problem solves the K-set agreement problem. Moreover, $(1, k)$-SSA is k-set agreement, while $(s, 1)$-SSA is s-simultaneous consensus.

3.1.2 The Failure Detector Class $Z_{s,k}$

Definition A failure detector of the class $Z_{s,k}$ provides each process p_i with two arrays denoted $qr_i[1..s]$ and $\ell d_i[1..s]$. Intuitively, $qr_i[z]$ and $\ell d_i[z]$, $1 \leq z \leq s$, denote, with respect to the index z, the current quorum and the current leader of p_i, respectively. $Z_{s,k}$ is defined by the following properties, where $qr_i^\tau[z]$ and $\ell d_i^\tau[z]$ denote the value of $qr_i[z]$ and $\ell d_i[z]$ at time τ.

- Safety property. $\forall\, z \in [1..s]$:
  - Quorum intersection property (QI).
    $\forall\, i_1, ..., i_{k+1} \in \Pi,\ \forall\, \tau_1, ..., \tau_{k+1} : \exists h, \ell \in [1..k+1] :\ (h \neq \ell) \wedge (qr_{i_h}^{\tau_h}[z] \cap qr_{i_\ell}^{\tau_\ell}[z] \neq \emptyset)$.
  - Leader validity property(LV). $\forall\, \tau,\ \forall\, i : \ell d_i^\tau[z] \in \Pi$.
- Liveness property. $\exists z \in [1..s]$:
  - Quorum liveness property (QL). $\forall\, i \in C : \exists\, \tau : \forall\, \tau' \geq \tau : qr_i^{\tau'}[z] \subseteq C$.
  - Eventual leadership property (EL). $\exists \ell \in C :\ \forall i \in C :$
    $\big[\forall\, \tau : \exists\, \tau', \tau'' \geq \tau : (qr_i^{\tau'}[z] \cap qr_\ell^{\tau''}[z] \neq \emptyset)\big] \ \Rightarrow\ \big[\exists\, \tau : \forall\, \tau' \geq \tau : (\ell d_i^{\tau'}[z] = \ell)\big]$.

The quorum intersection property states that, for any $z \in \{1, ..., s\}$, there are two quorum values that intersect in any set of $k+1$ quorum values, each taken at any time. The leader validity property states that the leader domain is the set of processes.

While the safety properties concern all the entries of the arrays $qr_i[1..s]$ and $\ell d_i[1..s]$, the liveness properties are only on a single of these entries, say z. The quorum liveness property states that there is a finite time after which all quorum values (appearing in $qr_i[z]$ for every $i \in C$) contain only correct processes. The eventual leader liveness property involves only the quorum values taken by the entries $qr_i[z]$, for every $i \in C$. Hence, it relates these quorum values with the eventual leader values in the local variables $\ell d_i[z]$ at each correct process p_i. More precisely, it states that there is a correct process p_ℓ such that, for any correct process p_i whose quorum $qr_i[z]$ intersects infinitely often with the quorum $qr_\ell[z]$ of p_ℓ (left part of the implication), p_ℓ becomes eventually the permanent leader of p_i (saved in $\ell d_i[z]$, right part of the implication).

Notation Let $Z(Q)_{s,k}$ denote the quorum part of $Z_{s,k}$ (defined by the properties QI and QL). Similarly, let $Z(L)_{s,k}$ denote the leader part of $Z_{s,k}$ (defined by the properties LV and EL where the quorum part brings no information on failures, which means that we have then $\forall\, i, \forall\, z,\ \forall \tau :$ $qr_i^\tau[z] = \Pi$).

Particular cases This part shows the generality of $Z_{s,k}$ by displaying failure detector classes that have appeared in the literature.

- $s = k = 1$: $Z_{1,1} = (\Sigma, \Omega)$ as defined in [19, 24].
- $Z_{s,k}$ is weaker than the failure detector $\Pi\Sigma_{k,s}$ introduced in [67].
- $Z_{s,1}$ is the failure detector W_s as defined in [17].
- $Z(Q)_{1,k}$ is the quorum failure detector Σ_k introduced in [12].

- $Z(Q)_{s,1}$ is the failure detector $V\Sigma_s$ introduced in [17] where it is shown that (a) $V\Sigma_s$ (i.e., $Z(Q)_{s,1}$) is necessary to solve the s-simultaneous consensus problem, and (b) $(V\Sigma_s, \overline{\Omega}_s) \preceq W_s (= Z_{s,1}) \preceq (V\Sigma_s, \Omega)$.

- $Z(Q)_{1,n-1}$ is the quorum failure detector $\mathcal{L}$ introduced in [26]. This follows from the fact that $Z(Q)_{1,n-1}$ is Σ_{n-1} and the equivalence between Σ_{n-1} and $\mathcal{L}$ established in [66].

- $Z(L)_{s,1}$ is the failure detector $\overline{\Omega}_k$ introduced in [75, 96] and used in [35].

3.2 A $Z_{s,k}$-based Algorithm for the (s, k)-SSA problem

This section presents a simple algorithm that solves the (s, k)-SSA problem in $\mathcal{AMP}[Z_{s,k}]$. This algorithm consists in the concurrent execution of s algorithms, each solving an instance of the k-set agreement problem (i.e., an instance of the $(1, k)$-SSA problem). This algorithm is based on an underlying abstraction (object) called alpha_k.

3.2.1 The Abstraction alpha_k

Historical perspective The abstraction alpha_k originates from a similar abstraction (denoted alpha) introduced in [37] (see also [80]) to capture the safety property of the consensus problem in message-passing systems. It has then been generalized to capture the safety property of k-set agreement in (a) read/write systems in [88], and (b) message-passing systems in [17, 67].

Definition We consider here the alpha_k object that we have introduced in [67]. Let $\perp$ be a default value that cannot be proposed by processes. alpha_k is an object, initialized to $\perp$, that may store up to k different values proposed by processes. It is an abstraction (object) that provides processes with a single operation denoted $\mathsf{alpha\_propose}_k(r, v)$ (where r is a round number and v a proposed value), which returns a value to the invoking process. The round number plays the role of a logical time that allows identifying the $\mathsf{alpha\_propose}_k()$ invocations. It is assumed that distinct processes use different round numbers and successive invocations by the same process use increasing sequence numbers. alpha_k is an *abortable* object in the sense that $\mathsf{alpha\_propose}_k()$ invocations are allowed to return the default value $\perp$ (i.e., abort) in specific concurrency-related circumstances (as defined from the obligation property, see below). More precisely, the alpha_k objects used in this chapter are defined by the following specification in which the obligation property takes explicitly into account the fact that these objects are used in the system model $\mathcal{AMP}[Z_{1,k}]$ (which is strictly stronger than $\mathcal{AMP}$). The properties defining such an alpha_k object are the following.

- Termination. Any invocation of $\mathsf{alpha\_propose}_k()$ by a correct process terminates.

- Validity. If $\mathsf{alpha\_propose}_k(r, v)$ returns $v' \neq \perp$, then $\mathsf{alpha\_propose}_k(r', v')$ has been invoked with $r' \leq r$.

- Quasi-agreement. At most k values different from the default value $\perp$ can be returned by the invocations of $\mathsf{alpha\_propose}_k()$.

- Obligation. (As $s = 1$, $qr_i[1]$ is denoted qr_i.) p_ℓ being a correct process, let $Q(\ell, \tau) = \{i \in C \mid \forall\, \tau_i, \tau_\ell \geq \tau :\ qr_i^{\tau_i} \cap qr_\ell^{\tau_\ell} = \emptyset\}$. If, after some finite time τ, (a) only p_ℓ and processes in $Q(\ell, \tau)$ invoke $\mathsf{alpha\_propose}_k()$ and (b) p_ℓ invokes $\mathsf{alpha\_propose}_k()$ infinitely often, then at least one invocation issued by p_ℓ returns a non-$\perp$ value.

Differently from the obligation property stated in [16, 37, 88] the previous obligation property is $Z_{s,k}$-aware (or more precisely $Z_{1,k}$-aware, i.e., Σ_k-aware). This obligation property allows concurrent invocations of $\mathsf{alpha\_propose}_k()$ to return non-$\perp$ values as soon as the quorums of the invoking processes do not intersect during these invocations. An algorithm implementing the previous alpha_k object in $\mathcal{AMP}[\Sigma_k]$ is described in [67].

3.2.2 A Base Algorithm for the $(1, k)$-SSA Problem (k-Set Agreement)

Algorithm 1 is a very simple algorithm solving the k-set agreement problem in $\mathcal{AMP}[Z_{1,k}]$. A process p_i invokes $\mathsf{ssa\_propose}_{1,k}\ (v_i)$ where v_i is the value it proposes. It decides a value v when it executes the statement $\mathsf{return}\ (v)$ which terminates its invocation. The local variable r_i is the local round number (as it is easy to see, each process uses increasing round numbers and no two different processes use the same round numbers).

A process loops until it decides. If during a loop iteration p_i is such that $\ell d_i = i$ (where ℓd_i denotes $\ell d_i[1]$, the single leader entry locally output at p_i by $Z_{1,k}$), p_i invokes the underlying alpha_k distributed object (denoted $ALPHA_k$) to try to deposit its value v_i into it (the success depends on the concurrency and quorums pattern). If a non-$\perp$ value is returned by this invocation, p_i broadcasts it and decides (execution of $\mathsf{return}()$). If it has not yet decided, a process decides when it receives a DECISION() message (lines 5-6 implement a reliable broadcast).

```
operation ssa_propose_{1,k} (v_i):
(1)  dec_i ← ⊥; r_i ← i;
(2)  while (dec_i = ⊥) do
(3)    if (ℓd_i = i) then dec_i ← ALPHA_k.alpha_propose_k(r_i, v_i); r_i ← r_i + n end if
(4)  end while;
(5)  for each j ∈ {1, ..., n} do send DECISION(dec_i) to p_j end for; return (v).

when DECISION(v) is received:
(6)  for each j ∈ {1, ..., n} do send DECISION(dec_i) to p_j end for; return (v).
```

Algorithm 1: k-Set agreement in $\mathcal{AMP}[Z_{1,k}]$ (code for p_i)

Theorem 9 *Algorithm 1 solves the k-set agreement problem in $\mathcal{AMP}[Z_{1,k}]$.*

Proof Validity and agreement properties of k-set agreement. Let us first observe that, due to the test of line 2, the default value $\perp$ cannot be decided. The fact that a decided value is a proposed value follows then from the validity of the underlying alpha_k object. Similarly, the fact that at most k non-$\perp$ values are decided follows directly from the quasi-agreement property of the underlying alpha_k object.

Termination property of k-set agreement. It follows from lines 5 and 6 that, at soon as a process decides (invokes $\mathsf{return}()$) each correct process eventually delivers the same DECISION(v) message and decides (if not yet done). The proof is by contradiction: assuming that no process decides, we show that at least one correct process executes line 5 (and consequently, all correct processes decide).

Let p_ℓ be the correct process that appears in the definition of the eventual leadership property of $Z_{1,k}$. It follows from the definition of p_ℓ that we eventually have forever $\ell d_\ell = \ell$.

Let R_ℓ be the (possibly empty) set of identities of the processes p_j (with $j \neq \ell$) such that we have $\ell d_j = j$ infinitely often. It follows from the contrapositive of the eventual leadership property of $Z_{1,k}$ that there is a time τ_{R_ℓ} such that $\forall\ j \in R_\ell, \forall\ \tau 1, \tau 2 \geq \tau_{R_\ell}$: $qr_j^{\tau 1} \cap qr_\ell^{\tau 2} = \emptyset$, from which we conclude that $R_\ell \subseteq Q(\ell, \tau_{R_\ell})$ (this is the set defined in the obligation property defining an alpha_k object).

Let us notice that, due to test of line 3, there is a finite time τ_a after which the only processes that invoke $\mathsf{alpha\_propose}_k()$ are the processes in $R_\ell \cup \{\ell\}$. Moreover (as by the contradiction assumption no process decides) it follows that, after τ_a, p_ℓ invokes $\mathsf{alpha\_propose}_k()$ infinitely often. Let τ_b be a time greater than $\max(\tau_{R_\ell}, \tau_a)$ from which we have $R_\ell \subseteq Q(\ell, \tau_{R_\ell}) \subseteq Q(\ell, \tau_b)$.

As after τ_b (a) only processes in $R_\ell \cup \{\ell\}$ invoke $\mathsf{alpha\_propose}_k()$, (b) p_ℓ invokes the operation $\mathsf{alpha\_propose}_k()$ infinitely often, and (c) $R_\ell \subseteq Q(\ell, \tau_b)$, we conclude from the obligation property of the alpha_k object that at least one invocation of p_ℓ returns a value $v \neq \perp$ and consequently sends the message DECISION(v) to all the processes. This contradicts the fact that no process decides and concludes the proof of the theorem. $\square_{Theorem\ 9}$

3.2.3 An Algorithm for the (s, k)-SSA Problem

A simple algorithm solving the (s, k)-SSA problem can be easily obtained by launching s concurrent instances of algorithm 1, the zth instance ($1 \leq z \leq s$) relying, at each process p_i, on the components $qr_i[z]$ and $\ell d_i[z]$ of $\mathcal{AMP}[Z_{s,k}]$. A process decides the value returned by the first of the s instances that locally terminates. Hence, it decides the pair (c, v) where c is its first deciding instance and v the value it decides in that instance. As there are s instances of algorithm 1 and at most k values can be decided in each of them, it follows that at most $K = sk$ different values can be decided. Moreover, as there is at least one instance z such that the failure detector outputs $\ell d_i[z]$ at each correct process p_i converge to the same correct process, it follows that the correct processes decide (if not done before) in at least one of the s instances of algorithm 1.

3.3 $Z(Q)_{s,k}$ is Necessary to Solve the (s, k)-SSA problem

This section shows that $Z(Q)_{s,k}$ is necessary to solve the (s, k)-SSA problem as soon as we are looking for a failure detector-based solution. To that end, given an algorithm A that solves the (s, k)-SSA problem in $\mathcal{AMP}[FD]$, this section presents an algorithm that emulates the output of $Z(Q)_{s,k}$, namely an array $qr_i[1..s]$ at each process p_i, which satisfies the properties QI and QL. This means that it is possible to build $Z(Q)_{s,k}$ from any failure detector FD that can solve the (s, k)-SSA problem.

According to the usual terminology, $Z(Q)_{s,k}$ is *extracted* from the FD-based algorithm A. This extraction is a generalization of the algorithm introduced in [12], which extracts Σ_k from any failure detector-based algorithm that solves the k-set agreement problem.

The Extraction Algorithm Each process p_i participates in several executions of the algorithm A. S being a set of processes, A^S denotes the execution of A in which exactly the processes of S participate. In this execution, each process of S either decides, blocks forever, or crashes. So the execution of the extraction algorithm is composed of $2^n - 1$ executions of A.

The behavior of each process p_i is described in algorithm 2. The internal statements of the tasks $T1$ and $T5$, and the tasks $T2$-$T4$ are locally executed in mutual exclusion. The local array $Q_i[1..s]$ is initialized to $[\Pi, \ldots, \Pi]$. The aim of $Q_i[c]$ is to contain all the sets S such that a value has been decided in the cth instance of the k-set agreement of the execution of A^S.

Initially, each process p_i proposes its identity i to all the instances of A in which it participates. To that end it invokes $A^S.\mathsf{ssa\_propose}_{s,k}(i)$ for each set S such that $i \in S$ ($\mathsf{ssa\_propose}_{s,k}()$ is the operation associated with each instance of the (s, k)-SSA problem). When it decides in the cth k-set agreement of A^S (task $T3$), p_i adds the set S to $Q_i[c]$ and informs each other process p_j, which includes S in $Q_j[c]$ when it learns it (task $T4$).

Init: $Q_i[1,\ldots,s] \leftarrow [\Pi,\ldots,\Pi]$; $queue_i \leftarrow \langle 1,\ldots,n\rangle$;
for each $S \subseteq \Pi$ **such that**$(i \in S)$ **do** A^S.ssa_propose$_{s,k}(i)$ **end for**; activate the tasks $T1$ to $T5$.

Task $T1$: **repeat periodically** send ALIVE(i) to each p_j such that $j \in \Pi \setminus \{i\}$ **end repeat**.

Task $T2$: **when** ALIVE(j) **is received**: move j at the head of $queue_i$.

Task $T3$: **when** $(c,-)$ **is decided by** p_i **in the** cth k-set agreement instance of A^S:
$Q_i[c] \leftarrow Q_i[c] \cup \{S\}$; send DECISION$(c,S)$ to each p_j such that $j \in \Pi \setminus \{i\}$.

Task $T4$: **when** DECISION(c,S) **is received**: $Q_i[c] \leftarrow Q_i[c] \cup \{S\}$.

Task $T5$: **repeat forever**
for each $c \in \{1,...,s\}$ **do**
$min\_rank_i \leftarrow \min\{\max\{rank(queue_i,j), j \in S\}, S \in Q_i[c]\}$;
$qr_i[c] \leftarrow$ any $S_{min} \in Q_i[c]$ such that $\max\{rank(queue_i,j), j \in S_{min}\} = min\_rank_i$
end for;
end repeat.

Algorithm 2: Extracting $Z(Q)_{s,k}$ from a failure detector-based algorithm A solving the (s,k)-SSA problem

Each alive process p_i sends periodically messages ALIVE(i) (task $T1$) to inform the other processes that it is alive. When it receives a message ALIVE(j) (task $T2$), a process p_i moves j at the head of its local queue (denoted $queue_i$) which always contains all process identities. It follows that the identities of all the correct processes eventually precede in this queue the identities of all the faulty processes. (Initially, each queue $queue_i$ contains all process identities, in any order.)

$T5$ is a task whose aim is to repeatedly compute the current value of $qr_i[1..s]$. It uses the function $rank(queue_i, j)$ which returns the current rank of p_j in the queue $queue_i$. The value of $qr_i[c]$ is computed as follows. It is the "first set of $Q_i[c]$ with respect to $queue_i$" (i.e., with respect to the processes which are currently seen as being alive). This is captured with the help of the local variable $minrank_i$. As an example Let $Q_i[c] = \{\{3,4,9\},\{2,3,8\},\{4,7\},\{1,2,3,4,5,6,7,8,9\}\}$, and $queue_i = \langle 4,8,3,2,7,5,9,1,6\rangle$. We have then $minrank = 4$, and $S_{min} = \{2,3,8\}$. This set of identities is the first set of $Q_i[c]$ with respect to $queue_i$ because each of the other sets $\{3,4,9\}$, $\{4,7\}$, or $\{1,2,3,4,5,6,7,8,9\}$, includes an element (9, 7, and 6, respectively) that appears in $queue_i$ after all the elements of $\{2,3,8\}$ (in case several sets are "first", any of them can be selected).

Theorem 10 *Given any algorithm A that solves the (s,k)-SSA problem in $\mathcal{AMP}[FD]$, the extraction algorithm described in Figure 2 is a wait-free construction of a failure detector $Z(Q)_{s,k}$.*

Proof Proof of the quorum liveness property. We have to show that there is an entry z such that, after some finite time and for each $i \in C$, $qr_i[z]$ contains only correct processes. Let us consider the execution of A^C. As all the processes in C are correct (definition), A^C terminates. It then follows from the tasks $T3$ and $T4$ that there is an entry z such the set $Q_i[z]$ of each correct process eventually contains the set C.

Moreover, as each correct process sends forever messages ALIVE() (task $T1$), it follows that the identities of the correct processes appear in queue $queue_i$ before the identities of the faulty processes (task $T2$). It then follows from task $T5$ (which computes the "first set of $Q_i[z]$ with respect to $queue_i$") that, after some finite time, $qr_i[z]$ contains only correct processes.

Proof of the quorum safety property. We have to show that, for each entry $z \in \{1,...,s\}$, we have $\forall\, i_1,...,i_{k+1} \in \Pi,\ \forall\, \tau_1,...,\tau_{k+1} : \exists h,\ell \in [1..k+1] :\ (h \neq \ell) \wedge (qr_{i_h}^{\tau_h}[z] \cap qr_{i_\ell}^{\tau_\ell}[z] \neq \emptyset)$.

The proof is by contradiction.

Let us assume that there exist a family $(i_m)_{1\leq m\leq k+1}$ of $k+1$ (not necessarily distinct) process identities, a family $(\tau_m)_{1\leq m\leq k+1}$ of time instants and an integer $z \in \{1,\ldots,s\}$ such that for all $m_1, m_2 \in \{1,\ldots,k+1\}$ $qr_{i_{m_1}}^{\tau_{m_1}}[z] \cap qr_{i_{m_2}}^{\tau_{m_2}}[z] = \emptyset$. Let S_m denote the set $qr_{i_m}^{\tau_m}[z]$. Let us remark that the previous non-intersection assumption and the fact that no set S_m is empty imply that, for each $m \in \{1,\ldots,k+1\}$, a process $p_{j_m} \in S_m$ decided a value v_m in the zth k-set agreement of $\mathcal{A}^{S_m}$ (no set S_m can have the initial value Π since it would intersect any other one). As the sets $S_m, 1 \leq m \leq k+1$ are pairwise disjoint, there is an execution of $\mathcal{A}$ in which the set of participants is $\bigcup_{m\in\{1,\ldots,k+1\}} S_m$ that is indistinguishable from the considered execution of $\mathcal{A}^{S_m}$ from the point of view of each p_{j_m} (just consider that the messages between processes of S_{m_1} and S_{m_2} are delayed and received only after all processes decide). It follows that in this execution each process p_{j_m} also decides v_m in the zth instance of the k-set agreement. It then follows from the agreement property of the k-set agreement problem that $|\{v_m, 1 \leq m \leq k+1\}| \leq k$. Hence, there exist $m_1, m_2 \in \{1,\ldots,k+1\}$ such that $v_{m_1} = v_{m_2}$. But, since the values decided in the zth instance of the k-set agreement problems involved in both $\mathcal{A}^{S_{m_1}}$ and $\mathcal{A}^{S_{m_2}}$ are identities of participating processes (this follows from the validity property of the k-set agreement problem), it follows that $v_{m_1} = v_{m_2} \in S_{m_1} \cap S_{m_2}$. This contradicts the initial assumption, which concludes the proof of the quorum intersection property of $Z(Q)_{s,k}$. $\square_{Theorem\ 10}$

3.4 The Structure of Generalized (s,k)-SSA problems

This section studies the mathematical structure of the family of (s,k)-SSA problems for $sk = K$. To that end, it first introduces a straightforward generalization of this family and then shows that this generalized family can be represented by a directed graph where an arrow from A to B means that the problem B can be solved from a black box solving the problem A, while the opposite is impossible. Given such a pair of problems (A,B), this section also associates with this pair a failure detector that is necessary to solve A and a failure detector that is sufficient to solve B.

3.4.1 The Generalized Asymmetric $\{k_1,...,k_s\}$-SSA Problem

While the (s,k)-SSA problem is a symmetric problem which consists in s simultaneous instances of the k-set agreement problem, a simple generalization consists in considering an asymmetric version made up of s simultaneous instances of possibly different set agreement problems, namely the k_1-set agreement problem, the k_2-set agreement problem, etc., and the k_s-set agreement problem. Hence, among the proposed values, at most $K = \Sigma_{x=1}^{s} k_x$ different values are decided.

This asymmetric version is denoted $\{k_1,...,k_s\}$-SSA where $\{k_1,...,k_s\}$ is a multiset[1]. The particular instance where $k_1 = \cdots = k_s = k$ is the symmetric (s,k)-SSA problem. As permuting the integers k_x does not change the problem, we consider the canonical notation where $k_1 \geq k_2 \geq \ldots \geq k_s \geq 1$.

3.4.2 Associating a Graph with the Generalized $\{k_1,...,k_s\}$-SSA Problems

Graph definition Given an integer K and starting from the source vertex labeled with the multiset $\{1,...,1\}$ (K times the integer 1), let us define a graph denoted $G(K)$ as follows. Given a vertex labeled $\{k_1,...,k_s\}$ (initially, $s = K$ and $k_1 = \cdots = k_K = 1$), we add all possible vertices of $s-1$ elements labeled $\{k'_1,...,k'_{s-1}\}$ and directed edges from $\{k_1,...,k_s\}$ to each vertex

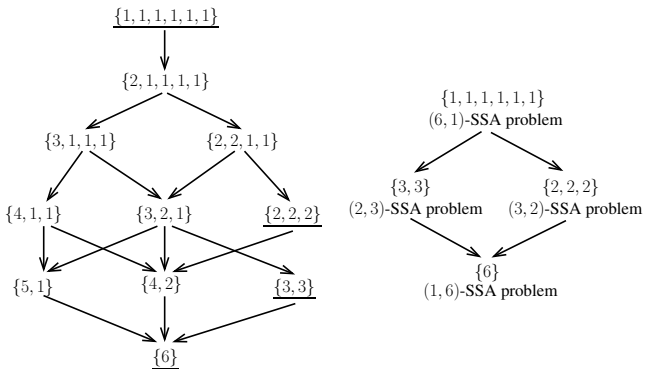
[1]The set notation is used to represent a multiset. A multiset is a set in which several elements can have the same value. As an example, $\{1,2,1,1,3\}$ is a multiset of 5 elements. Hence, the multisets $\{1,2,1,1,3\}$ and $\{2,1,3\}$ are different (while $\{1,2,1,1,3\} = \{2,1,3\}$ from a set point of view).

$\{k'_1, ..., k'_{s-1}\}$ defined as follows. Any pair of elements k_x, k_y of the multiset $\{k_1, ..., k_s\}$ gives rise to a vertex labeled by the multiset $\{k'_1, ..., k'_{s-1}\}$ such that

$$\{k'_1, ..., k'_{s-1}\} \ = \ \{k_1, ..., k_s\} \setminus \{k_x, k_y\} \cup \{k_x + k_y\}.$$

Then, the construction process is recursively repeated until we arrive at a sink node composed of a single element labeled $\{K\}$.

An example of the graph for $K = 6$ is given on the right. The labels corresponding to symmetric instances ((s,k)-SSA problems) are underlined. The graph (lattice) on the right side of the figure considers only the symmetric problem instances.



Meaning of the graph As we will see in Section 3.4.4, given an integer K, this graph describes the computability hierarchy linking all the $\{k_1, k_2, ...\}$-SSA agreement problems such that $k_1 + k_2 + \cdots = K$. Let the label A of a vertex denote both the vertex itself and the associated agreement problem. An edge from a vertex A to a vertex B means that (a) given an algorithm that solves the problem A it is possible to solve the problem B, while (b) the opposite is impossible.

Lemma 7 *$G(K)$ is cycle-free.*

Proof The proof is an immediate consequence of the fact that the successors of any vertex labeled by a multiset of size s are multisets of size $s-1$. $\square_{Theorem\ 7}$

As we will see in Lemma 8, the following predicate P characterizes (with the existence of a function f) the pairs of vertices connected by a path in $G(K)$.

Definition Let $\{k_1, ..., k_s\}$ and $\{k'_1, ..., k'_{s'}\}$ be any pair of vertices of $G(K)$.

$$P\big(\{k_1, ..., k_s\}, \{k'_1, ..., k'_{s'}\}\big) \stackrel{def}{=}$$
$$\exists f : \{1, \ldots, s\} \to \{1, \ldots, s'\} \text{ s.t. } \forall y \in \{1, \ldots, s'\} \ : k_y \ = \ \textstyle\sum_{x \in f^{-1}(y)} k_x.$$

Lemma 8 *$\big(\exists$ a path from $\{k_1, ..., k_s\}$ to $\{k'_1, ..., k'_{s'}\}\big) \Leftrightarrow \big(P(\{k_1, ..., k_s\}, \{k'_1, ..., k'_{s'}\})\big)$.*

Proof Direction $\Rightarrow$.
Let $\{a_1^1, \ldots, a_s^1\} = \{k_1, ..., k_s\}, ..., \{a_1^z, \ldots, a_{s-z+1}^z\}, ..., \{a_1^{s-s'+1}, \ldots, a_{s'}^{s-s'+1}\} = \{k'_1, ..., k'_{s'}\}$ be a path from $\{k_1, ..., k_s\}$ to $\{k'_1, ..., k'_{s'}\}$ in $G(K)$. (Hence, all the edges $(\{a_1^z, \ldots, a_{s-z+1}^z\}$, $\{a_1^{z+1}, \ldots, a_{s-z}^{z+1}\})$, $z \in \{1 \leq z \leq s - s'\}$, belong to $G(K)$.)

For each $z \in \{1, \ldots, s - s'\}$, it follows from the definition of $G(K)$ that there is a pair (x_z, y_z) of distinct integers in $\{1, \ldots, s - z + 1\}$ such that $a_{x_z}^z \geq a_{y_z}^z$, $x_z < y_z$ (w.l.o.g.), and $\{a_1^{z+1}, \ldots, a_{s-z}^{z+1}\} = \{a_1^z, \ldots, a_{s-z+1}^z\} \setminus \{a_{x_z}^z, a_{y_z}^z\} \cup \{a_{x_z}^z + a_{y_z}^z\}$. Let f_z be the function that associates with each $x \in \{1, \ldots, s-z+1\}$ (i) x if $a_x^z > a_{x_z}^z + a_{y_z}^z$, (ii) $x+1$ if $a_{x_z}^z + a_{y_z}^z \geq a_x^z$ and $x < x_z$, (iii) x if $a_{x_z}^z + a_{y_z}^z \geq a_x^z$ and $x_z < x < y_z$, (iv) $x - 1$ if $a_{x_z}^z + a_{y_z}^z \geq a_x^z$ and $y_z < x$, and (v) the index w_0 of the first occurrence of $a_{x_z}^z + a_{y_z}^z$ in $\{a_1^{z+1}, \ldots, a_{s-z}^{z+1}\}$ if $x \in \{x_z, y_z\}$. (The aim of these cases is to place the element $a_{x_z}^z + a_{y_z}^z$ at its right place in the multiset $\{a_1^{z+1}, \ldots, a_{s-z}^{z+1}\}$.)

By construction, for all $y \in \{1, \ldots, s - z\} \setminus \{w_0\}$, $f^{-1}(y)$ contains a single value and $a_{f^{-1}(y)}^z = a_y^{z+1}$, while $f^{-1}(w_0) = \{x_z, y_z\}$. Since $a_{x_z}^z + a_{y_z}^z = a_{w_0}^{z+1}$, $P(\{a_1^z, \ldots, a_{s-z+1}^z\}$,

$\{a_1^{z+1}, \ldots, a_{s-z}^{z+1}\})$ is satisfied. Let f be $f_z \circ f_{z-1} \circ \cdots \circ f_1$, since the property verified by f is stable by composition, the path from $\{k_1, ..., k_s\}$ to $\{k'_1, ..., k'_{s'}\}$ ensures that the property $P(\{k_1, \ldots, k_s\}, \{k'_1, ..., k'_{s'}\})$ is satisfied.

Direction $\Leftarrow$.
Let $(\{k_1, ..., k_s\}, \{k'_1, ..., k'_{s'}\})$ be a pair of vertices of $G(K)$ that satisfies the predicate P. Two cases are possible: (i) $s = s'$ and then f is the identity function (or just a reordering of the values appearing multiple times) and $\{k_1, ..., k_s\} = \{k'_1, \ldots k'_{s'}\}$ or (ii) $s > s'$.

In the case (i), there is a path (of length zero) from any node to itself. In the case (ii), consider the function f defined in the predicate P. Since $s > s'$, f is not injective (one-to-one), and there are two distinct integers x_1, y_1 such that $f(x_1) = f(y_1)$. Let then $\{a_1^2, \ldots, a_{s-1}^2\}$ be $\{k_1, ..., k_s\} \setminus \{k_{x_1}, k_{y_1}\} \cup \{k_{x_1+y_1}\}$. If f is injective from $\{1, \ldots, s\} \setminus \{y_1\}$ to $\{1, \ldots, s'\}$, since according to P it is surjective (onto), it follows that $s' = s - 1$ and that $\{a_1^2, \ldots, a_{s-1}^2\} = \{k'_1, ..., k'_{s'}\}$ and there is a path in $G(K)$ from $\{k_1, ..., k_s\}$ to $\{k'_1, ..., k'_{s'}\}$. If f is not injective from $\{1, \ldots, s\} \setminus \{y_1\}$ to $\{1, \ldots, s'\}$, then there exist two distinct integers x_2, y_2 in $\{1, \ldots, s\} \setminus \{y_1\}$ such that $f(x_2) = f(y_2)$ (consider that $y_2 \neq x_1$, w.l.o.g.). Let then $\{a_1^3, \ldots, a_{s-2}^3\}$ be (i) $\{k_1, ..., k_s\} \setminus \{k_{x_1}, k_{y_1}, k_{x_2}, k_{y_2}\} \cup \{k_{x_1} + k_{y_1}, k_{x_2} + k_{y_2}\}$ if $x_1 \neq x_2$, or (ii) $\{k_1, ..., k_s\} \setminus \{k_{x_1}, k_{y_1}, k_{y_2}\} \cup \{k_{x_1} + k_{y_1} + k_{y_2}\}$ if $x_2 = x_1$. If f is injective from $\{1, \ldots, s\} \setminus \{y_1, y_2\}$ to $\{1, \ldots, s'\}$ then, as it is also surjective, $s' = s - 2$, $\{a_1^3, \ldots, a_{s-2}^3\} = \{k'_1, ..., k'_{s'}\}$ and there is a path of length 2 in $G(K)$ from $\{k_1, ..., k_s\}$ to $\{k'_1, ..., k'_{s'}\}$. If f is not injective from $\{1, \ldots, s\} \setminus \{y_1, y_2\}$ to $\{1, \ldots, s'\}$ then let us choose x_3, y_3 in $\{1, \ldots, s\} \setminus \{y_1, y_2\}$ such that $f(x_3) = f(y_3)$ and $y_3 \notin \{x_1, x_2\}$ and continue the construction. After $s - s'$ iteration steps, we have $|\{1, \ldots, s\} \setminus \{y_1, \ldots, y_{s-s'}\}| = s'$, and the vertices which have been traversed belong to a path of $G(K)$ starting at $\{k_1, ..., k_s\}$ and ending at $\{k'_1, ..., k'_{s'}\}$. $\square_{Lemma\ 8}$

Theorem 11 *The transitive closure of $G(K)$ is a partial order.*

Proof The theorem follows from the fact that the relation captured by the predicate P is anti-symmetric (Lemma 7) and transitive (Lemma 8). $\square_{Theorem\ 11}$

3.4.3 Associated Generalized Failure Detector $GZ_{k_1,...,k_s}$

The failure detector $Z_{s,k}$ is implicitly tailored for the symmetric (s, k)-SSA problem. A simple generalization allows to extend it to obtain an "equivalent" failure detector suited to asymmetric problems.

As $Z_{s,k}$, this generalized failure detector, denoted $GZ_{k_1,...,k_s}$, provides each process p_i with an array $qr_i[1..s]$ and an array $\ell d_i[1..s]$. It differs from $Z_{s,k}$ in the constraint imposed by the quorum intersection property that is now specific to each entry $z \in \{1, ..., s\}$. More explicitly, QI is replaced by the property GQI defined as follows

- Quorum intersection property (GQI). $\forall\, z \in [1..s]$:
  $\forall\, i_1, ..., i_{k_z+1} \in \Pi,\ \forall\, \tau_1, ..., \tau_{k_z+1} : \exists h, \ell \in [1..k_z + 1] :\ (h \neq \ell) \wedge (qr_{i_h}^{\tau_h}[z] \cap qr_{i_\ell}^{\tau_\ell}[z] \neq \emptyset)$.

The other properties –leader validity (LV), quorum liveness (QL), and eventual leader liveness (EL)– remain unchanged. It is easy to see, that $GZ_{k_1,...,k_s}$ boils down to $Z_{s,k}$ when $k_1 = \cdots = k_s = k$.

Let $GZ(Q)_{k_1,...,k_s}$ denote the quorum part of $GZ_{k_1,...,k_s}$ (properties GQI and QL). The proof of the following theorem is a simple extension of the proof of Theorem 10. It is left to the reader.

Theorem 12 *Given any algorithm A that solves the $\{k_1, ..., k_s\}$-SSA problem in the model $\mathcal{AMP}[FD]$, the extraction algorithm described in Figure 2 is a wait-free construction of a failure detector $GZ(Q)_{k_1,...,k_s}$.*

3.4.4 A Hierarchy of Agreement Problems

Problem hierarchy Let $\mathcal{AMP}[X]$ denote the asynchronous message-passing model in which any number of processes may crash ($\mathcal{AMP}$) enriched with an algorithm that solves the problem X.

Given the the message-passing model $\mathcal{AMP}$, a problem A is *stronger* than a problem B (denoted $A \succeq B$) if B can be solved in $\mathcal{AMP}[A]$ (we also say that B is *weaker* than A, denoted $B \preceq A$). Moreover, A is *strictly stronger* than B (denoted $A \succ B$) if $A \succeq B$ and $\neg(B \succeq A)$ (A cannot be solved in $\mathcal{AMP}[B]$).

Lemma 9 $P\big(\{k_1, ..., k_s\}, \{k'_1, ..., k'_{s'}\}\big) \Rightarrow \big(\{k_1, \ldots, k_s\}\text{-SSA} \succeq \{k'_1, \ldots, k'_{s'}\}\text{-SSA}\big)$.

Proof Let us consider an algorithm $\mathcal{A}$ that solves the $\{k_1, \ldots, k_s\}$-SSA problem. To solve the $\{k'_1, \ldots, k'_{s'}\}$-SSA problem it is sufficient that each process p_i executes $\mathcal{A}$ until it decides a pair (x, v) and then outputs the pair $(f(x), v)$ as the decided value for the $\{k'_1, \ldots, k'_{s'}\}$-SSA problem (where f is the function appearing in P).

The validity and termination properties of the obtained algorithm follow directly from those of $\mathcal{A}$. Moreover, according to the agreement property satisfied by $\mathcal{A}$, at most k_x different pairs $(x, -)$ can be decided by the underlying algorithm. It then follows from the definition of f that for any $y \in \{1, \ldots, s'\}$, at most $\sum_{x \in f^{-1}(y)} k_x = k_y$ distinct pairs $(y, -)$ are decided in the simulated solution of the $\{k'_1, \ldots, k'_{s'}\}$-SSA problem, and the agreement property of the $\{k'_1, \ldots, k'_{s'}\}$-SSA is satisfied. $\square_{Lemma\ 9}$

Lemma 10 $\big(\{k_1, \ldots, k_s\}\text{-SSA} \succeq \{k'_1, \ldots, k'_{s'}\}\text{-SSA}\big) \Rightarrow P\big(\{k_1, \ldots, k_s\}, \{k'_1, \ldots, k'_{s'}\}\big)$ *for any K such that $n > K \geq 2$.*

Proof The principle and structure of the proof are as follows.

- The proof considers first the failure detector $GZ_{k_1,...,k_s}$, which is sufficient to solve the $\{k_1, ..., k_s\}$-SSA problem (the (s, k)-SSA algorithm described in Section 3.2 can be easily adapted to solve the $\{k_1, ..., k_s\}$-SSA problem in $\mathcal{AMP}[GZ_{k_1,...,k_s}]$).
- As $\{k_1, \ldots, k_s\}\text{-SSA} \succeq \{k'_1, \ldots, k'_s\}\text{-SSA}$, it is possible to solve the $\{k'_1, \ldots, k'_{s'}\}$-SSA problem in the model $\mathcal{AMP}[GZ_{k_1,...,k_s}]$. Moreover, as $GZ(Q)_{k'_1,...,k'_{s'}}$ is necessary to solve the $\{k'_1, \ldots, k'_{s'}\}$-SSA problem (Theorem 12), it is possible to simulate $GZ(Q)_{k'_1,...,k'_{s'}}$ in $\mathcal{AMP}[GZ_{k_1,...,k_s}]$.
- The proof constructs such a simulation and shows that this simulation allows to define a function f such that the predicate $P\big(\{k_1, \ldots, k_s\}, \{k'_1, \ldots, k'_{s'}\}\big)$ is satisfied, from which the lemma follows.

Assuming $n > K \geq 2$, and $\mathcal{A}$ being an algorithm that simulates $GZ(Q)_{k'_1,...,k'_{s'}}$ in the model $\mathcal{AMP}[GZ_{k_1,...,k_s}]$, let $QR_i[1, \ldots, s']$ denote the outputs of the simulated quorum at process p_i. Moreover, for each process $p_i, 1 \leq i \leq K+1$ and for each $x \in \{1, \ldots, s\}$, let α_i^x be an execution of $\mathcal{A}$ in which the only process to take steps is p_i and the underlying failure detector $GZ_{k_1,...,k_s}$ always outputs (locally at each p_i) $qr_i[x] = \{i\}$, $ld_i[x] = i$, and $\forall x' \neq x, qr_i[x'] = \{1, \ldots, K+1\}$ and $ld_i[x'] = i$ (let us notice that these outputs comply with the definition of the underlying failure

detector $GZ_{k_1,...,k_s}$). According to the quorum liveness property of the constructed failure detector $GZ(Q)_{k'_1,...,k'_{s'}}$, there exist in α_i^x an instant τ_i^x and an index c_i^x such that $\forall \tau \geq \tau_i^x \ : \ QR_i[c_i^x] = \{i\}$.

Considering the solo executions $\{\alpha_i^x\}_{1 \leq i \leq K+1}$, one can build, for any subset of K distinct processes such that $\{i_1, \ldots, i_K\} \subset \{1, \ldots, K+1\}$, an execution of $\mathcal{A}$ denoted

$$\alpha_{i_1 \cdots i_K} = \alpha_{i_1}^1 \cdot \alpha_{i_2}^1 \cdots \alpha_{i_{k_1}}^1, \alpha_{i_{k_1+1}}^2 \cdots \alpha_{i_{k_1+k_2}}^2, \ldots, \alpha_{i_{K-k_s+1}}^s \cdots \alpha_{i_K}^s$$

as follows where $W(x)$ denotes the integer interval $[1 + k_1 + \cdots + k_{x-1}..k_1 + \cdots + k_x]$:

- All processes $p_i, i > K+1$, crash before taking any step,
- All message receptions are delayed after time $\tau\_max = \max\{\tau_i^x$ such that $i \in \{1, ..., K+1\}, x \in \{1, ..., s\}\}$,[2]
- Let us observe that $\alpha_{i_1 \cdots i_K}$ contains the prefix of the solo execution $\alpha_{i_w}^x$ of the process p_{i_w} if and only if $w \in W(x)$.

  For each $x \in \{1, \ldots, s\}$ and for each $w \in W(x)$, the outputs at p_{i_w} of the underlying failure detector $GZ_{k_1,...,k_s}$ are the same as in $\alpha_{i_w}^x$ until τ_max. After this instant, for each $i \in \{i_1, \ldots, i_K\}$ and for $x \in \{1, \ldots, s\}$, $qr_i[x] = \{i_1, \ldots, i_K\}$ and $ld_i[x] = i_1$.

Let us remark that the outputs of the underlying failure detector $GZ_{k_1,...,k_s}$ are valid in this execution, and that, for each $x \in \{1, \ldots, s\}$ and each $p_{i_w}, w \in W(x)$, the execution $\alpha_{i_1 \cdots i_K}$ is indistinguishable from $\alpha_{i_w}^x$ until τ_max from the point of view of p_{i_w}. It follows that for each $x \in \{1, \ldots, s\}$ and each $p_{i_w}, w \in W(x)$, we have $QR_{i_w}[c_{i_w}^x] = \{i_w\}$ at time τ_max in $\alpha_{i_1 \cdots i_K}$. It follows from this observation and the quorum intersection property of the failure detector $GZ(Q)_{k'_1,...,k'_{s'}}$ built by $\mathcal{A}$ (this property has to be preserved on each quorum index $y \in \{1, \ldots, s'\}$) that $\{c_{i_1}^1, \ldots, c_{i_{k_1}}^1, \ldots, c_{i_{K-k_s+1}}^s, \ldots, c_{i_K}^s\}$ is a multiset of K elements where for each $y \in \{1, \ldots, s'\}$, the value y appears k'_y times (and those are the only values of the elements of this multiset).

Let us now consider the execution $\alpha_{\sigma(1) \cdots \sigma(K)}$ where σ is any permutation of $\{1, \ldots, K\}$. According to the previous discussion, any process $p_{i_{\sigma(w)}}, w \in \{1, \ldots, K\}$ can be replaced by p_{K+1} in the execution without changing the multiset $\{c_{i_{\sigma(1)}}^1, \ldots, c_{i_{\sigma(k_1)}}^1, \ldots, c_{i_{\sigma(K-k_s+1)}}^s, \ldots, c_{i_{\sigma(K)}}^s\}$. It then follows that $\forall x \in \{1, \ldots, s\}, \forall w \in \{1, \ldots, K\} : (\sigma(w) \in W(x) \Rightarrow c_{i_{\sigma(w)}}^x = c_{K+1}^x)$. As σ can be any permutation, it follows that $\forall x \in \{1, \ldots, s\}, \forall w \in \{1, \ldots, K\} : c_w^x = c_{K+1}^x$. Moreover this implies that $\forall x \in \{1, \ldots, s\}, \forall i, j \in \{1, \ldots, K+1\} : c_i^x = c_j^x$.

Let us finally consider the function f which, with each $x \in \{1, \ldots, s\}$ associates c_1^x (which, as just shown, is equal to c_i^x for all $i \in \{1, \ldots, K\}$). Since, $\{c_{i_1}^1, \ldots, c_{i_{k_1}}^1, \ldots, c_{i_{K-k_s+1}}^s, \ldots, c_{i_K}^s\}$ contains k_x times $f(x)$ for each $x \in \{1, \ldots, s\}$ (and only these values), it follows from the multiset equality above that $\forall y \in \{1, \ldots, s'\} : \Sigma_{x \in f^{-1}(y)} k_x = k_y$, which ends the proof of the lemma. $\square_{Lemma\ 10}$

Theorem 13 $\big(\{k_1, \ldots, k_s\}\text{-SSA} \succeq \{k'_1, \ldots, k'_{s'}\}\text{-SSA}\big) \Leftrightarrow P\big(\{k_1, \ldots, k_s\}, \{k'_1, \ldots, k'_{s'}\}\big)$ *for any* K *such that* $n > K \geq 2$.

Proof The proof follows directly from Lemma 9 and Lemma 10. $\square_{Theorem\ 13}$

[2] This is the only place where the "asynchronous message-passing" assumption is used. If communication was through atomic read/write registers, this message asynchrony would not exist and the lemma would not hold.

Theorem 14 *The relation $\succ$ on generalized-*SSA *problems is a partial order.*

Proof The proof follows from Theorem 11 ($G(K)$ is a partial order), Lemma 8 (all paths in ($G(K)$ are characterized by P), and Theorem 13 (which relates P and $\succ$). $\square_{Theorem\ 14}$

The next corollary follows from the observation that, for any $K > 1$, the K-set agreement problem is a sink vertex in the directed graph $G(K)$.

Corollary 2 *The weakest failure detector for the K-set agreement problem does not allow to solve any $\{k_1, \ldots, k_s\}$-SSA problem such that $s > 1$ and $k_1 + \cdots + k_s = K$.*

3.4.5 The Lattice of Symmetric SSA Problems

As seen before, a symmetric vertex is a vertex $\{k_1, ..., k_s\}$ such that $k_1 = \ldots = k_s = k$. Let $SG(K)$ denote the graph whose vertices are the symmetric vertices of $G(K)$, and there is an edge from (s_x, k_x) to (s_y, k_y) iff there is a path in $G(K)$ from the vertex $\{k_x, ..., k_x\}$ (k_x appearing s_x times) to the vertex $\{k_y, ..., k_y\}$ (k_y appearing s_y times) and no path connecting these vertices passes through a symmetric vertex. As an example, $SG(6)$ is given in Section 3.4.2.

Theorem 15 *For any K, $SG(K)$ is a lattice.*

Proof Let (s_x, k_x) and (s_y, k_y) be two pairs of integers such that $k_x < k_y$ and $s_x k_x = s_y k_y = K$. It follows from the definition of $SG(K)$ that there is an edge from (s_x, k_x) to (s_y, k_y) iff $k_y = k_x p$ where p is prime (if p is not prime there is a symmetric vertex on a path from (s_x, k_x) to (s_y, k_y), and if k_y/k_x is not an integer, there is no path from (s_x, k_x) to (s_y, k_y)). It follows that we can associate with any pair of pairs (s_x, k_x) and (s_y, k_y) such that $s_x k_x = s_y k_y = K$,

- among its ancestors, the vertex (s_z, k_z) in $SG(K)$ where $s_z k_z = K$ and $k_z = gcd(k_x, k_y)$, and
- among its successors, the vertex (s'_z, k'_z) in $SG(K)$ where $s'_z k'_z = K$ and $k'_z = lcm(k_x, k_y)$.

As the greatest common denominator and the least common multiple of any pair of integers are unique, it follows that $SG(K)$ is a lattice. $\square_{Theorem\ 15}$

The next corollary follows from the previous theorem.

Corollary 3 *Let (s_1, k_1) and (s_2, k_2) be two different pairs of integers such that $s_1 k_1 = s_2 k_2$, and none of k_1 and k_2 divides the other one. The symmetric (s_1, k_1)-*SSA *and (s_2, k_2)-*SSA *problems are incomparable in $\mathcal{AMP}$.*

As far as agreement problems are concerned, this shows a strong difference between the message-passing model and the read/write model. In the read/write model, (s_1, k_1)-SSA and (s_2, k_2)-SSA are the same problem (they are both equivalent to the K-simultaneous problem which is itself equivalent to the K-set agreement problem, where $K = s_1 k_1 = s_2 k_2$).

3.5 Concluding Remarks on the Relations Between k-Set Agreement, s-Simultaneous Consensus and their Generalizations

In this chapter, we showed that the k-set agreement and s-simultaneous consensus problems (with $k = s$) in $\mathcal{AMP}$ can be viewed as parts of a broader strict hierarchy made of hybrid simultaneous set-agreement problems. We presented a family of failure detectors that provides, for each of these

problems, a failure detector that is sufficient to solve it (along with the corresponding algorithm) and one that is necessary. The structure formed by the computability relations between these problems has been extensively described.

We hope that this work can be instrumental in the quest for the weakest failure detector to solve the k-set agreement in $\mathcal{AMP}$. Indeed, it provides insights on how this problem is different from the simultaneous consensus in that model, while both are equivalent in $\mathcal{ARW}$ in which the associated weakest failure detector is known [75, 96].

Chapter 4

Universal Construction from k-Simultaneous Consensus Objects and Registers

This chapter presents an extension of the k-universal construction presented in [34]. It follows a modular approach allowing to build universal constructions, based on simultaneous consensus objects, with various liveness conditions for both the operations of the processes and the simulated objects. A way to avoid the use of the simultaneous consensus objects when there is no contention is also proposed. This work has been published in [87].

Section 4.1 introduces the notion of universal construction and discusses the differences between our proposal and the original construction of [34]. Section 4.2 defines the considered model, the adopt-commit object used by our algorithms and the properties of a wait-free linearizable universal construction. Section 4.3 presents the original algorithm of [34] and discusses its limitations and how it can be improved. Section 4.4 describes our base obstruction-free algorithm and proves its validity, Section 4.5 proposes incremental improvements to obtain wait-freedom for the operations of the processes as well as the possibility to use stronger simultaneous consensus objects in order to guarantee the progress of several of the simulated objects. Finally Section 4.6 concludes the chapter.

4.1 Universal Construction and its Generalization

Asynchronous crash-prone read/write systems and the notion of a universal construction A fundamental problem encountered in these asynchronous wait-free shared memory systems consists in implementing any object, defined by a sequential specification, in such a way that the object behaves reliably despite process crashes.

Several progress conditions have been proposed for concurrent objects. The most extensively studied, and strongest condition, is wait-freedom. Wait-freedom guarantees that $every$ process will always be able to complete its pending operations in a finite number of its own steps [38]. Thus, a $wait\text{-}free$ implementation of an object guarantees that an invocation of an object operation may fail to terminate only when the invoking process crashes. The non-blocking progress condition (sometimes called lock-freedom) guarantees that $some$ process will always be able to complete its pending operations in a finite number of its own steps [46]. Obstruction-freedom guarantees that a process will be able to complete its pending operations in a finite number of its own steps, if all the other processes "hold still" long enough [40]. Obstruction-freedom does not guarantee progress under contention.

It has been shown in [31, 38, 57] that the design of a general algorithm implementing any object defined by a sequential specification and satisfying the wait-freedom progress condition, is impossible in $\mathcal{ARW}$. Thus, in order to be able to implement any such object, the model has to be enriched with basic objects whose computational power is stronger than atomic read/write registers [38].

Objects that can be used, together with registers, to implement any other object which satisfies a given progress condition PC, are called universal objects with respect to PC. Previous work provided algorithms, called $universal\ constructions$, based on universal objects, that transform sequential specifications of arbitrary objects into wait-free concurrent implementations of the same objects. It is shown in [38] that the *consensus* object is universal with respect to wait-freedom. A consensus object allows all the correct processes to reach a common decision based on their initial inputs. A consensus object is used in a universal construction to allow processes to agree –despite concurrency and failures– on a total order on the operations they invoke on the constructed object.

In addition to the universal construction of [38], several other wait-free universal constructions were proposed, which address additional properties. As an example, a universal construction is presented in [28], where "processessection-aa:GG11-construction operating on different parts of an implemented object do not interfere with each other by accessing common base objects". Other additional properties have been addressed in [6, 30].

Universal construction for k objects An interesting question introduced in [34] by Gafni and Guerraoui is the following: what happens if, when considering the design of a universal construction, k-simultaneous consensus objects are considered instead of consensus objects? The authors claim that k-simultaneous consensus objects are *k-universal* in the sense that they allow to implement k deterministic concurrent objects, each defined by a sequential specification "with the guarantee that $at\ least\ one$ machine remains highly available to all processes" [34]. In their paper, Gafni and Guerraoui focus on the replication of k state machines. They present a k-universal construction, based on the replication –at every process– of each of the k state machines. This construction is presented in Section 4.3.

Contributions wrt Generalized Universality [34] This work is focused on *distributed universality*, namely it presents a very general universal construction for a set of n processes that access k concurrent objects, each defined by a sequential specification on total operations. An operation on an object is "total" if, when executed alone, it always returns [46]. This construction is based on a generalization of the k-simultaneous consensus object (see below). The noteworthy features of this construction are the following.

- At least ℓ among the k objects progress forever, $1 \leq \ell \leq k$. This means that an infinite number of operations is applied to each of these ℓ objects. This set of ℓ objects is not predetermined, and depends on the execution.

- The progress condition associated with the processes is wait-freedom. That is, a process that does not crash executes an infinite number of operations on each object that progresses forever.

- An object stops progressing when no more operations are applied to it. The construction guarantees that, when an object stops progressing, all its copies (one at each process) stop in the same state.

- The construction is *contention-aware*. This means that the overhead introduced by using synchronization objects other than atomic read/write registers is eliminated when there is

no contention during the execution of an operation (i.e., interval contention). In the absence of contention, a process completes its operations by accessing only read/write registers[1]. Algorithms which satisfy the contention-awareness property have been previously presented in [10, 58, 61, 93].

- The construction is *generous* with respect to *obstruction-freedom*.This means that each process is able to complete its pending operations on all the k objects each time all the other processes hold still long enough. That is, if once and again all the processes except one hold still long enough, then all the k objects, and not just ℓ objects, are guaranteed to always progress.

According to these properties, this new universal construction is consequently called a *wait-free contention-aware obstruction-free-generous* (k, ℓ)*-universal construction*. Differently, the universal construction presented in [34] is a $(k, 1)$-universal construction and is neither contention-aware, nor generous with respect to obstruction-freedom. Moreover, this construction suffers from the following limitations: (a) it does not satisfy wait-freedom progress, but only non-blocking progress (i.e., infinite progress is guaranteed for only one process); (b) in some scenarios, an operation that has been invoked by a process can (incorrectly) be applied twice, instead of just once; and (c) the last state of the copies (one per process) of an object on which no more operations are being executed can be different at distinct processes. While issue (b) can be fixed (see Section 4.3), we do not see how to modify the construction from [34] to overcome drawback (c).

When considering the special case $k = \ell = 1$, Herlihy's construction is wait-free $(1, 1)$-universal [38], but differently from ours, it does not satisfy the contention-awareness property.

To ensure the progress of at least ℓ of the k implemented objects, the proposed construction uses a new synchronization object, that we call (k, ℓ)-simultaneous consensus object, which is a simple generalization of the k-simultaneous consensus object. This object type is such that its $(k, 1)$ instance is equivalent to k-simultaneous consensus, while its (k, k) instance is equivalent to consensus. Thus, when added to the basic $\mathcal{ARW}$ system model, (k, ℓ)-simultaneous consensus objects add computational power. We show that (k, ℓ)-simultaneous consensus objects are both *necessary and sufficient* to ensure that at least ℓ among the k objects progress forever.

From a software engineering point of view, the proposed (k, ℓ)-universal construction is built in a modular way. First a non-blocking $(k, 1)$-universal construction, using k-simultaneous consensus objects and atomic registers, is designed. Interestingly, its design principles are different from the other universal constructions we are aware of. Then, this basic construction is extended to obtain a contention-aware $(k, 1)$-universal construction, and then a wait-free contention-aware $(k, 1)$-universal construction. Finally, assuming that the system is enriched with (k, ℓ)-simultaneous consensus objects, $1 \leq \ell \leq k$, instead of k-simultaneous consensus objects, we obtain a contention-aware wait-free (k, ℓ)-universal construction. During the modular construction, we make sure that the universal construction implemented at each stage is also generous with respect to obstruction-freedom.

4.2 Basic and Enriched Models, and Wait-free Linearizable Implementation

4.2.1 Basic read/write model and enriched model

The basic model considered in this chapter is $\mathcal{ARW}$. In addition to atomic read/write registers [51], two other types of objects are used. The first type, the $adopt\text{-}commit$ object (see fur-

[1]Let us recall that, in *worst case* scenarios, hardware operations such as compare&swap() can be $1000\times$ more expensive that read or write.

ther), does not add computational power, but provides processes with a higher abstraction level. The other type, the k-simultaneous consensus defined in Introduction 1.4.3 adds computational power to the basic system model $\mathcal{ARW}$.

Adopt-commit object The adopt-commit object has been introduced in [33]. An adopt-commit object is a one-shot object that provides the processes with a single operation denoted propose(). This operation takes a value as an input parameter, and returns a pair (tag, v). The behavior of an adopt-commit object is formally defined as follows:

- Validity.
  - Result domain. Any returned pair (tag, v) is such that (a) v has been proposed by a process and (b) $tag \in \{commit,\ adopt\}$.
  - No-conflicting values. If a process p_i invokes propose(v) and returns before any other process p_j has invoked propose(v') with $v' \neq v$, then only the pair $(commit, v)$ can be returned.
- Agreement. If a process returns $(commit, v)$, the only pairs that are allowed to be returned are $(commit, v)$ and $(adopt, v)$.
- Termination. An invocation of propose() by a correct process always terminates.

Let us notice that it follows from the "no-conflicting values" property that, if a single value v is proposed, then only the pair $(commit, v)$ can be returned. Adopt-commit objects can be wait-free implemented in $\mathcal{ARW}$ (e.g., [33, 80]). Hence, they provide processes with a higher abstraction level than read/write registers.

4.2.2 Correct object implementation

Let us consider n processes that access k concurrent objects, each defined by a deterministic sequential specification. The sequence of operations that p_i wants to apply to an object m, $1 \leq m \leq k$, is stored in the local infinite list $my\_list_i[m]$, which can be defined statically or dynamically (in that case, the next operation issued by a process p_i on an object m, can be determined from p_i's view of the global state). It is assumed that the processes are well-formed: no process invokes a new operation on an object m before its previous operation on m has terminated.

Wait-free linearizable implementation An implementation of an object m by n processes is wait-free linearizable if it satisfies the following properties.

- Validity. If an operation op is executed on object m, then $op \in \cup_{1 \leq i \leq n} my\_list_i[m]$, and all the operations of $my\_list_i[m]$ which precede op have been applied to object m.
- No-duplication. Any operation op on object m invoked by a process is applied at most once to m. We assume that all the invoked operations are unique.
- Consistency. Any n-process execution produced by the implementation is linearizable [46].
- Termination (wait-freedom). If a process does not crash, it executes an infinite number of operations on at least one object.

Weaker progress conditions In some cases, the following two weaker progress conditions are considered.

- The *non-blocking* progress condition [46] guarantees that there is at least one process that executes an infinite number of operations on at least one object.
- The *obstruction-freedom* progress condition [40] guarantees that any correct process can complete its operations if it executes in isolation for a long enough period (i.e., there is a long enough period during which the other processes stop progressing).

4.3 Gafni and Guerraoui's Non-blocking k-Universal Construction

4.3.1 Gafni and Guerraoui's construction

This section presents Gafni and Guerraoui's generalized non-blocking k-universal construction introduced in [34], and denoted GG in the following. To make reading easier, we use the same variable names as in the construction presented in Figure 4.2 for local and shared objects that have the same meaning in both constructions. The objects considered in GG are deterministic state machines, and "operations" are accordingly called "commands".

Principle The algorithm GG is based on local replication, namely, the only shared objects are the control objects $kSC[1..]$ and $AC[1..][1..k]$. Each process p_i manages a copy of every state machine m, denoted $machine_i[m]$, which contains the last state of machine m as known by p_i. The invocation by p_i of $machine_i[m].\mathsf{execute}(c)$ applies the command c to its local copy of machine m.

As explained in [34], the use of a naive strategy to update local copies of states machines, makes possible the following bad scenario. During a round r, a process p_1 executes a command $c1$ on its copy of machine $m1$, while a process p_2 executes a command $c2$ on a different machine $m2$. Then, during round $r+1$, p_1 executes a command $c2'$ on the machine $m2$ without having executed first $c2$ on its copy of $m2$. This bad behavior is prevented from occurring in [34] by a combined used of adopt-commit objects and an appropriate marking mechanism. When a process p_i applies a command c to its local copy of a machine m, it has necessarily received the pair $(commit, c)$ from the adopt-commit object associated with the current round, and consequently the other processes have received $(commit, c)$ or $(adopt, c)$. The process p_i attaches then to its next command for machine m, namely $oper_i[m]$, the indication that $oper_i[m]$ has to be applied to m after c, so that no process executes $oper_i[m]$ without having previously executed c.

Algorithm As before, $my\_list_i[m]$ defines the list of commands that p_i wants to apply to the machine m. Moreover, $my\_list_i[m].\mathsf{first}()$ sets the read head to point to the first element of this list and returns its value; $my\_list_i[m].\mathsf{current}()$ returns the command under the read head; finally, $my\_list_i[m].\mathsf{next}()$ advances the read head before returning the command pointed to by the read head.

The algorithm is described in Figure 4.1. As the algorithm of Figure 4.2, it is round-based and has the same first four lines. When a process p_i enters a new asynchronous round (line 1), it first executes line 2-4, which are the lines involving the k-simultaneous consensus object and the adopt-commit object associated with the current round r.

After the execution of these lines, for $1 \leq m \leq k$, $(tag_i[m], ac\_op_i[m])$ contains the command that p_i has to consider for the machine m. For each of them it does the following. First, if $ac\_op_i[m]$ is marked "to be executed after" $oper_i[m]$, p_i applies $oper_i[m]$ to $machine_i[m]$ (lines 6-8). Then, if $tag_i[m] = adopt$, p_i adopts $ac\_op_i[m]$ as its next proposal for machine m

```
r_i ← 0;
for each m ∈ {1, ..., k} do
    machine_i[m] ← initial state of the state machine m; oper_i[m] ← my_list_i[m].first()
end for.

repeat forever
(1)  r_i ← r_i + 1;
(2)  (ksc_obj, ksc_op) ← kSC[r_i].propose(oper_i[1..k]);
(3)  (tag_i[ksc_obj], ac_op_i[ksc_obj]) ← AC[r_i][ksc_obj].propose(ksc_op);
(4)  for each m ∈ {1, ..., k} \ {ksc_obj} do
                (tag_i[m], ac_op_i[m]) ← AC[r_i][m].propose(oper_i[m]) end for;
(5)  for each m ∈ {1, ..., k} do
(6)             if (ac_op_i[m] is marked "to_be_executed_after" oper_i[m])
(7)               then machine_i[m].execute(oper_i[m])
(8)             end if;
(9)             if (tag_i[m] = adopt)
(10)              then oper_i[m] ← ac_op_i[m]
(11)              else machine_i[m].execute(ac_op_i[m]);        % tag_i[m] = commit %
(12)                   if ac_op_i[m] = my_list_i[m].current()
(13)                     then oper_i[m] ← my_list_i[m].next()
(14)                     else oper_i[m] ← my_list_i[m].current()
(15)                   end if;
(16)                   mark oper_i[m] "to_be_executed_after" ac_op_i[m]
(17)            end if
(18) end for
end repeat.
```

Figure 4.1: Gafni-Guerraoui's generalized universality non-blocking algorithm (code of p_i) [34]

(lines 9-10). Otherwise, $tag_i[m] = commit$. In this case p_i first applies $ac\_op_i[m]$ to its local copy of the machine m (line 11). Then, if $ac\_op_i[m]$ was a command it has issued, p_i computes its next proposal $oper_i[m]$ for the machine m (lines 12-15). Finally, to prevent the bad behavior previously described, it attaches to $oper_i[m]$ the fact that this command cannot be applied to any copy of the machine m before the command $ac\_op_i[m]$ (line 16).

4.3.2 Discussion: Gafni-Guerraoui's construction revisited

The GG algorithm has two main drawbacks. First, it does not prevent a process from executing twice the same command on a given machine. Second, it is possible that, when a state machine stops progressing, it stops in different states at different processes. While the first issue can be easily fixed (see below), the second seems more difficult to work around.

Let us consider the following execution of the GG algorithm (Figure 4.1). During some round r, a process p_i applies a command c to its local copy of the machine m (hence, p_i obtained $(commit, c)$ from $AC[r][m]$, and each other process has obtained either $(commit, c)$ or $(adopt, c)$). It follows from line 16 that p_i marks its next command on m ($c' = oper_i[m]$) "to be executed after c". Let us consider now two distinct scenarios for the round $r + 1$.

Scenario 1. It is possible that all the processes, except p_i, have received $(adopt, c)$ during the round r and propose c to $AC[r + 1][m]$. Moreover, according to the specification of an adopt-commit object, nothing prevents $AC[r + 1][m]$ from outputting $(commit, c)$ at all the processes. In this case p_i will execute the command c twice on $machine_i[m]$. This erroneous behavior can be easily fixed by adding the following filtering after line 8:

if ($oper_i[m]$ is marked "to_be_executed_after"$ac\_op_i[m]$)

then do not execute the lines 9-17
end if.

This filtering amounts to check if the command $ac\_op_i[m]$ has already been locally executed. The fact that $ac\_op_i[m]$ has been previously committed is encoded in $oper_i[m]$ by the marking mechanism.

Scenario 2. Let us again consider the round $r+1$, and consider the possible case where the pair $(m, -)$ is not output by $kSC[r+1]$ (let us remember that $kSC[r+1]$ outputs one pair per process and globally at least one and at most k pairs). According to the specification of $AC[r+1][m]$, it is possible to have $(tag_j[m], ac\_op_j[m]) = (adopt, c)$ at any process $p_j \neq p_i$, and $(tag_i[m], ac\_op_i[m]) = (adopt, c')$ where c' is the new command that p_i wants to apply to the machine m. Hence, as far as m is concerned, all the processes execute the lines 9-10, and we are in the same configuration as at the end of round r. It follows that this can repeat forever. If it is the case, p_i has executed one more command on its local copy of machine m than the other processes. This means that state machine m stops progressing in different states at distinct processes.

4.4 A New Non-blocking k-Universal Construction

As mentioned in Section 4.1, the construction is done incrementally. In this section, we present and prove the correctness of a non-blocking k-universal construction, based on new design principles (as far as we know). This construction is built in the enriched model $\mathcal{ARW}[k\text{-}SC]$. In Section 4.5, we extend the construction, without requiring additional computational power, to obtain the contention-awareness property, and the wait-freedom progress condition (i.e., $each$ correct process can always execute and complete its operations on any object that progresses forever). Then (k, ℓ)-SC objects are introduced (which are a natural generalization of k-SC objects), and are used to design a (k, ℓ)-universal construction which ensures that least ℓ objects progress forever. In Section 4.5, we also show that (k, ℓ)-SC objects are necessary and sufficient to obtain a (k, ℓ)-universal construction.

4.4.1 A new non-blocking k-universal construction: data structures

The following objects are used by the construction. Identifiers with upper case letters are used for shared objects, while identifiers with lower case letters are used for local variables.

Shared objects

- $kSC[1..]$: infinite list of of k-simultaneous consensus objects; $kSC[r]$ is the object used at round r.

- $AC[1..][1..k]$: infinite list of vectors of k adopt-commit objects; $AC[r][m]$ is the adopt-commit object associated with the object m at round r.

- $GSTATE[1..n]$ is an array of single-writer/multi-readers atomic registers; $GSTATE[i]$ can be written only by p_i. Moreover, the register $GSTATE[i]$ is made up of an array with one entry per object, such that $GSTATE[i][m]$ is the sequence of operations that have been applied to the object m, as currently know by p_i; it is initialized to ϵ (the empty sequence).

Local variables at process p_i

- r_i: local round number (initialized to 0).
- $g\_state_i[1..n]$: array used to save the values read from $GSTATE[1..n]$.
- $oper_i[1..k]$: vector such that $oper_i[m]$ contains the operation that p_i is proposing to a k-SC object for the object m (as we will see in the algorithm, this operation was not necessarily issued by p_i).
- $my\_op_i[1..k]$: vector of operations such that $my\_op_i[m]$ is the last operation that p_i wants to apply to the object m (hence $my\_op_i[m] \in my\_list_i[m]$).
- $\ell\_hist_i[1..k]$: vector with one entry per object, such that $\ell\_hist_i[m]$ is the sequence of operations defining the history of object m, as known by p_i. Each $\ell\_hist_i[m]$ is initialized to ϵ. The function append() is used to add an element at the end of a sequence $\ell\_hist_i[m]$.
- $tag_i[1..k]$ and $ac\_op_i[1..k]$: arrays that, for each object m, are used to save the pairs $(tag, operation)$ returned by the invocation of $AC[r][m]$ of current round r.
- $output_i[1..k]$: vector such that $output_i[m]$ contains the result of the last operation invoked by p_i on the object m (this is the operation saved in $my\_op_i[m]$).

Without loss of generality, it is assumed that each object operation returns a result, which can be "ok" when there is no object-dependent result to be returned (as with the stack operation push() or the queue operation enqueue()).

4.4.2 A new non-blocking k-universal construction: algorithm

To simplify the presentation, it is assumed that each operation invocation is unique. This can be easily realized by associating an identity (process id, sequence number) with each operation invocation. In the following, the term "operation" is used as an abbreviation for "operation execution".

The function next() is used by a process p_i to access the sequence of operations $my\_list_i[m]$. The x-th invocation of $my\_list_i[m]$.next() returns the xth element of this list.

Initialization The algorithm implementing the k-universal construction is presented in Figure 4.2. For each object $m \in \{1, ..., k\}$, a process p_i initializes both the variables $my\_op_i[m]$ and $oper_i[m]$ to the first operation that it wants to apply to m. Process p_i then enters an infinite loop.

Repeat loop: using the round r objects $kSC[r]$ and $AC[r]$ (lines 1-4) After it has increased its round number, a process p_i invokes the k-simultaneous consensus object $kSC[r]$ to which it proposes the operation vector $oper_i[1..n]$, and from which it obtains the pair (ksc_obj, ksc_op); ksc_op is an operation proposed by some process for the object ksc_obj (line 2). Process p_i then invokes the adopt-commit object $AC[r][ksc\_obj]$ to which it proposes the operation output by $kSC[r]$ for the object ksc_op (line 3). Finally, for all the other objects $m \neq ksc\_obj$, p_i invokes the adopt-commit object $AC[r][m]$ to which it proposes $oper_i[m]$ (line 4). As already indicated, the tags and the commands defined by the vector of pairs output by the adopt-commit objects $AC[r]$ are saved in the vectors $tag_i[1..k]$ and $ac\_op_i[1..k]$, respectively. (While expressed differently, these four lines are the only part which is common to this construction and the one presented in [34].)

The aim of these lines is to implement a filtering mechanism such that (a) for each object, at most one operation can be committed at some processes, and (b) there is at least one object for which an operation is committed at some processes.

```
for each m ∈ {1,...,k} do
    my_op_i[m] ← my_list_i[m].next(); oper_i[m] ← my_op_i[m] end for.

repeat forever
(1)  r_i ← r_i + 1;
(2)  (ksc_obj, ksc_op) ← kSC[r_i].propose(oper_i[1..k]);
(3)  (tag_i[ksc_obj], ac_op_i[ksc_obj]) ← AC[r_i][ksc_obj].propose(ksc_op);
(4)  for each m ∈ {1,...,k} \ {ksc_obj} do
         (tag_i[m], ac_op_i[m]) ← AC[r_i][m].propose(oper_i[m]) end for;

(5)  for each j ∈ {1,...,n} do g_state_i[j] ← GSTATE[j] end for;
     % the read of each GSTATE[j] is atomic %
(6)  for each m ∈ {1,...,k} do
(7)    ℓ_hist_i[m] ← longest history of g_state_i[1..n][m] containing ℓ_hist_i[m];
(8)    if (my_op_i[m] ∈ ℓ_hist_i[m])                    % my operation was completed %
(9)          then output_i[m] ← compute_output(my_op_i[m], ℓ_hist_i[m]);
(10)               return {(m, my_op_i[m], output_i[m])} to the upper layer;
(11)               my_op_i[m] ← my_list[m].next()
(12)   end if
(13) end for;

(14) res ← ∅;
(15) for each m ∈ {1,...,k} do
(16)     if (ac_op_i[m] ∉ ℓ_hist_i[m])                  % operation was not completed %
(17)       then if (tag_i[m] = commit)                    % complete the operation %
(18)              then ℓ_hist_i[m] ← ℓ_hist_i[m].append(ac_op_i[m]);
(19)                   if (ac_op_i[m] = my_op_i[m])      % my operation was completed %
(20)                     then output_i[m] ← compute_output(ac_op_i[m], ℓ_hist_i[m]);
(21)                          res ← res ∪ {(m, my_op_i[m], output_i[m])};
(22)                          my_op_i[m] ← my_list[m].next()
(23)                   end if;
(24)                   oper_i[m] ← my_op_i[m]
(25)              else oper_i[m] ← ac_op_i[m]                 % tag_i[m] = adopt %
(26)            end if
(27)       else oper_i[m] ← my_op_i[m]                  % ac_op_i[m] ∈ ℓ_hist_i[m] %
(28)     end if
(29) end for;

(30) GSTATE[i] ← ℓ_hist_i[1..k];                  % globally update my current view %
(31) if (res ≠ ∅) then return res to the upper layer end if
end repeat.
```

Figure 4.2: Basic Non-Blocking Generalized $(k, 1)$-Universal Construction (code for p_i)

Repeat loop: returning local results (lines 5-13) Having used the additional power supplied by $kSC[r]$, a process p_i first obtains asynchronously the value of $GSTATE[1..n]$ (line 5) to learn an "as recent as possible" consistent global state, which is saved in $g\_state_i[1..n]$. Then, for each object m (lines 6-13), p_i computes the maximal local history of the object m which contains $\ell\_hist_i[m]$ (line 7). (Let us notice that $g\_state_i[i][m]$ is $\ell\_hist_i[m]$.) This corresponds to the longest history in the n histories $g\_state_i[1][m]$, ..., $g\_state_i[n][m]$ which contains $\ell\_hist_i[m]$. If there are several longest histories, they all are equal as we will see in the proof. If the last operation it has issued on m, namely $my\_op_i[m]$, belongs to this history (line 8), some process has executed this operation on its local copy of m. Process p_i computes then the corresponding output (line 9), locally returns the triple $(m, my\_op_i[m], output_i[m])$ (line 10), and defines its next local operation to apply to the object m (line 11).

The function compute_output(op, h) (used at lines 9 and 20) computes the result returned by

op applied to the state of the corresponding object m (this state is captured by the prefix of the history h of m ending just before the operation op).

Repeat loop: trying to progress on machines (lines 14-29) Then, for each object m, $1 \leq m \leq k$, p_i considers the operation $ac\_op_i[m]$. If this operation belongs to its local history $\ell\_hist_i[m]$ (the predicate of line 16 is then false), it has already been locally applied; p_i consequently assigns $my\_op_i[m]$ to $oper_i[m]$, where its next operation on the object m is saved (line 27).

If $ac\_op_i[m] \notin \ell\_hist_i[m]$ (line 16), the behavior of p_i depends on the fact that the tag of $ac\_op_i[m]$ is $commit$ or $adopt$. If the tag is $adopt$ (the predicate of line 17 is then false), p_i defines $ac\_op_i[m]$ as the next operation it will propose for the object m, which is saved in $oper_i[m]$ (line 25): it "adopts" $ac\_op_i[m]$. If the tag is $commit$ (line 17), p_i adds (applies) the operation $ac\_op_i[m]$ to its local history (line 18). Moreover, if $ac\_op_i[m]$ has been issued by p_i itself (i.e., $ac\_op_i[m] = my\_op_i[m]$, line 19), p_i computes the result locally returned by $ac\_op_i[m]$ (line 20), adds this result to the set of results res (line 21), defines its next local operation to apply to the object m (line 22). Finally, p_i assigns $my\_op_i[m]$ to $oper_i[m]$ (line 24).

Repeat loop: making public its progress (lines 30-31) Finally, p_i makes public its current local histories (one per object) by writing them in $GSTATE[i]$ (line 30), and returns local results if any (line 31). It then progresses to the next round.

4.4.3 A new non-blocking k-universal construction: proof

Lemma 11 $\forall\ i, m$: $(op \in GSTATE[i][m]) \Rightarrow (\exists\ j : op \in my\_list_j[m])$ *(i.e., if an operation op is applied to an object m, then op has been proposed by a process).*

Proof Before being written into $GSTATE[i][m]$ (line 31), an operation op is first appended to m's local history for the first time at line 18. It follows from lines 2-4 that this operation was proposed to an adopt-commit object by some process p_j in $oper_j[m]$. If $oper_j[m]$ was updated in the initialization phase, at line 24 or line 27, it is an operation of $my\_list_j[m]$. If $oper_j[m]$ was updated at line 25, it was proposed to an adopt-commit object by another process p_x, and (by a simple induction) the previous reasoning shows that this operation belongs then to some $my\_list_z[m]$. $\square_{Lemma\ 11}$

Lemma 12 $\forall\ i, j, m : (op \in my\_list_j[m]) \Rightarrow$ *(op appears at most once in* $GSTATE[i][m]$*) (i.e., an operation is executed at most once).*

Proof Suppose by contradiction that, at a given time and for an object m, there is a history $GSTATE[-][m]$ that contains twice the same operation op. Let p_i be the first process that wrote such a history with op appearing twice in $GSTATE[i][m]$, and let τ be the time instant at which p_i does it. Since $GSTATE[i][m]$ is written only at line 31 with the content of $\ell\_hist_i[m]$, p_i necessarily stored before τ a history containing twice op in $\ell\_hist_i[m]$. As $\ell\_hist_i[m]$ is initially empty, it does not contain twice op in the initial state of p_i. Since $\ell\_hist_i[m]$ is updated only at line 7 or line 18, p_i sets it to a history containing twice op at one of these lines. According to the predicate of line 16, p_i cannot append op to $\ell\_hist_i[m]$ at line 18 if op already appears in that sequence. It follows that p_i updates $\ell\_hist_i[m]$ before τ at line 7 with one of the longest local histories of m which contains op twice. Consequently, when p_i read (non-atomically) $GSTATE$ at line 5, it retrieved that history from one of the $GSTATE[j][m]$, also before τ. But this contradicts the fact that no process writes a history containing op twice before τ. It follows that no

history containing several times the same operation can ever be written into one of the registers $GSTATE[-][-]$. $\square_{Lemma\ 12}$

The sequence $(op_r^m)_{r\geq 1}$ of committed operations According to the specification of the adopt-commit object, for any round r and any object m there is at most one operation returned with the tag $commit$ by the object $AC[r][m]$ to some processes. Let op_r^m denote this unique operation if at least one process obtains a pair with the tag $commit$, and let op_r^m be $\perp$ if all the pairs returned by $AC[r][m]$ contain the tag $adopt$.

From the sequence $(op_r^m)_{r\geq 1}$ to the notion of valid histories Considering an execution of the algorithm of Figure 4.2, the following lemmas show that, for any process p_i and any object m, all the sequences of operations appearing in $\ell\_hist_i[m]$ are finite prefixes of a unique valid sequence depending only on the sequence $(op_r^m)_{r\geq 1}$ of committed operations.

More precisely, given a sequence $(op_r^m)_{r\geq 1}$, a history $(vh_x^m)_{1\leq x\leq xmax}$ is *valid* if it is equal to a sequence $(op_r^m)_{1\leq r\leq R}$ from which the $\perp$ values and the repetitions have been removed. More formally, $(vh_x^m)_{1\leq x\leq xmax}$ is valid if there is a round number R and a strictly increasing function $\sigma : \{1,\ldots,xmax\} \to \{1,\ldots,R\}$ such that for all x in $\{1,\ldots,xmax\}$: (a) $vh_x^m = op_{\sigma(x)}^m$, (b) $vh_x^m \neq \perp$, (c) for all x in $\{1,\ldots,xmax-1\}$: $vh_x^m \neq vh_{x+1}^m$, and (d) the sets $\{vh_1^m, ..., vh_{xmax}^m\}$ and $\{op_1^m, ..., op_R^m\} \setminus \{\perp\}$ are equal.

Let us remark that this definition has two consequences: (i) the value of R for which item (d) is verified defines unambiguously the sequence $(vh_x^m)_{1\leq x\leq xmax}$ (and accordingly this sequence is denoted $VH^m(R)$ in the following), and (ii) for any two valid histories $(vh_x^m)_{1\leq x\leq xmax1}$ and $(vh_x^m)_{1\leq x\leq xmax2}$, one is a prefix of the other.

Lemma 13 *For any process p_i and any object m, at any time the local history $\ell\_hist_i[m]$ is valid.*

Proof Let us suppose by contradiction that a process p_j updates $\ell\_hist_j[m]$ with a sequence that is not valid. Let p_i be the first process that writes an invalid sequence (denoted s) into its variable $\ell\_hist_i[m]$. Let ρ be the round and τ the time at which it does it.

Since p_i is the first process that writes s into its local history $\ell\_hist_i[m]$, it cannot do it at line 7 (this would imply that p_i retrieved s in some $g\_state_i[j][m]$ obtained from its previous non-atomic read of $GSTATE$ –line 5– implying that a process p_j would have written s into its local history $\ell\_hist_j[m]$ before τ). Consequently p_i writes s into $\ell\_hist_i[m]$ at line 18. It follows that the adopt-commit object $AC[\rho][m]$ returned to p_i the pair $(commit, op)$ (where op is the last operation in s) at line 3 or 4 during round ρ, hence, $op_\rho^m = op$.

Let us remind that, by assumption, before p_i appended op to $\ell\_hist_i[m]$ at line 18 of round ρ, $\ell\_hist_i[m]$ was valid; let s' denote that history. Moreover, as p_i executes line 18 of round ρ, it fulfilled the condition of line 16, hence we have $op \notin s'$. Let R_1 be the smallest (resp. R_2 the largest) round number R such that $s' = VH^m(R)$. It follows from the previous observation that $R_2 < \rho$, and from the definition of R_1, that $op_{R_1}^m \neq \perp$ ($op_{R_1}^m$ is the last operation appearing in $VH^m(R_1) = VH^m(R_2)$). Let us remark that, since s' is valid while s is not, there is necessarily a round number r such that $R_2 < r < \rho$, $op_r^m \neq \perp$ and $s' = VH^m(R_2) \neq VH^m(r)$ (intuitively, p_i "missed" a committed operation). Let r_0 be the smallest round number verifying these conditions. According to this definition, $op_{r_0}^m \neq op_{R_1}^m$.

Let us first show that $op_{r_0}^m \notin VH^m(R_1) = VH^m(R_2)$. Suppose by contradiction that it exists a round $r_1 < R_2$ such that $op_{r_1}^m = op_{r_0}^m$ and consider a process p_j executing round r_1. The proof boils down to show that such a process p_j cannot propose $op_{r_1}^m = op_{r_0}^m$ to a $kSC[r]$ object with $r > r_1 + 1$ before τ, which entails that this operation cannot be committed during round r_0 and

leads to a contradiction. If p_j commits $op^m_{r_1} = op^m_{r_0}$ during that round, then, after the execution of lines 16-28, it has op_{r_1} into its variable $\ell\_hist_i[m]$, has set its variable $oper_j[m]$ to a different operation and will never propose op_{r_1} further in the execution. If p_j adopts op_{r_1} during round r_1, then two cases are possible: (a) p_j returns from its invocation of $AC[r_1+1][m].\mathsf{propose}(-)$ before any process, which has committed op_{r_1} during round r_1, invokes $kSC[r_1+1][m].\mathsf{propose}(-)$, or (b) one of the processes that committed op_{r_1} during round r_1, invokes $kSC[r_1+1][m].\mathsf{propose}(-)$ before p_j returns from its invocation of $AC[r_1+1][m].\mathsf{propose}(-)$. In the case (a), according to the validity properties of the k-simultaneous consensus and adopt-commit objects, p_j commits op_{r_1} during round r_1+1 and, as before, will not propose this operation further in the execution since it appears in its local history. In the case (b), one of the processes that committed op_{r_1} during round r_1 wrote a history containing it before p_j executes line 5 of round r_1+1. If this happens before τ, then both this history and the history of p_j are valid, thus p_j adopts that history that strictly contains its own local history. It follows that p_j executes lines 16-28 of round r_1+1 with a history containing op_{r_1} and consequently never proposes this operation further in the execution. This ends the proof of the fact that $op^m_{r_0} \notin VH^m(R_1) = VH^m(R_2)$.

From the previous remark, it follows that, before τ, p_i never retrieves any history $VH^m(r)$ with $r \geq r_0$ during its non-atomic read of $GSTATE$ (or it would have set its variable $\ell\_hist_i[m]$ to one of these histories at line 7 and never reset it to s', since these histories contain $VH^m(r_0)$, and are consequently strictly longer than s').

Let us consider the execution of round r_0 by p_i (since p_i reaches line 18 of round $\rho > r_0$, this occurs). Let us suppose that p_i obtains the pair $(commit, op^m_{r_0})$ from $AC[r_0][m]$. As, (a) before τ, the values of $\ell\_hist_i[m]$ are valid (hence they can only increase), and (b) $op^m_{r_0} \notin VH^m(R_2)$, it follows that p_i appends $op^m_{r_0}$ to $\ell\_hist_i[m]$ at line 18 of round r_0, contradicting the fact that, just before τ, $\ell\_hist_i[m] = s' = VH^m(R_2)$. Consequently, according to the definition of r_0 and the specification of the adopt-commit object, $AC[r_0][m]$ returns $(adopt, op^m_{r_0})$ to p_i.

During round r_0, since $op^m_{r_0} \neq \perp$, all the processes that do not crash before obtain one of the two pairs $(adopt, op^m_{r_0})$ or $(commit, op^m_{r_0})$ from $AC[r_0][m]$. Let $\mathcal{C}$ denote the ones that obtain $(commit, op^m_{r_0})$, and $\mathcal{A}$ the ones that obtain $(adopt, op^m_{r_0})$. Among the processes of $\mathcal{A}$, some fulfill the condition of line 16 during round r_0, namely those which do not have $op^m_{r_0}$ in their local history. Let $\mathcal{A}_-$ denote this set of processes and let $\mathcal{A}_+$ be $\mathcal{A} \setminus \mathcal{A}_-$. As previously shown, p_i cannot have $op^m_{r_0}$ in $\ell\_hist_i[m]$ before τ; consequently $p_i \in \mathcal{A}_-$. Let μ be the first time at which a process of $\mathcal{C} \cup \mathcal{A}_+$ (the set of processes that have $op^m_{r_0}$ in their local histories at the end of round r_0) executes line 31 of round r_0. Let μ' be the first time at which one of these processes invokes $kSC[r_0+1][m].\mathsf{propose}(-)$ at round r_0+1. Let τ_i be the time at which p_i terminates its invocation of $AC[r_0+1][m].\mathsf{propose}(-)$, and τ'_i the time at which it terminates its read of line 5 during round r_0+1.

Let us remark that any process p_j of $\mathcal{A}_-$ (including p_i) starts round r_0+1 with $oper_j[m] = op^m_{r_0}$. It follows from the k-simultaneous consensus and adopt-commit specifications and the structure of the lines 2-4, that if $\tau_i < \mu'$ then p_i necessarily obtains the pair $(commit, op^m_{r_0})$ from $AC[r_0+1][m]$. As this happens before τ, $op^m_{r_0} \notin \ell\_hist_i[m]$ when p_i checks the condition of line 16, and it consequently appends op_{r_0} to $\ell\_hist_i[m]$ at line 18 of round r_0+1. This contradicts the fact that $s' = VH^m(R_2)$, except for the case $r_0+1 = \rho$. But, for $r_0+1 = \rho$, we should have $op^m_{r_0} = op^m_\rho = op$, and, by definition of r_0, s would be valid, which contradicts the fact that (due to the definition of s) it is not.

The only remaining case is thus $\mu' < \tau_i$, but since $\mu < \mu'$ and $\tau_i < \tau'_i$, it follows that $\mu < \tau'_i$ which implies that p_i obtains a valid history containing op_{r_0} during its read of $GSTATE$ at round r_0+1 and consequently updates $\ell\_hist_i[m]$ to one of these histories at line 7, thus before τ. This leads to a contradiction which concludes the proof of the lemma. $\square_{Lemma\ 13}$

The execution on an object m of an operation op, issued by a process p_i, starts when the process p_i proposes op to a k-simultaneous consensus object $kSC[-][m]$ for the first time (i.e., when p_i makes op public), and terminates when a set res including $(m, op, output[m])$ is returned by p_i at line 10 or line 31. The next lemma shows that any execution is linearizable.

Lemma 14 *The execution of an operation op issued by a process p_i on an object m can be linearized at the first time at which a process p_j writes into $GSTATE[j][m]$ a local history $\ell\_hist_j[m]$ such that $op \in \ell\_hist_j[m]$.*

Proof Let op be an operation applied on an object m and p_i be the process such that $op \in my\_list_i[m]$. Let us first show that op cannot appear in the local history $\ell\_hist_j[m]$ before being proposed by p_i to one of the k-simultaneous consensus objects $kSC[-][m]$. Let p_j be the first process that adds op to its local history $\ell\_hist_j[m]$ and τ the time at which this occurs. It follows that time τ cannot occur at line 7, but occurs when p_j executes line 18 when it appends op to $\ell\_hist_j[m]$ during some round r. Process p_j consequently obtained the pair $(commit, op)$ from the adopt-commit object $AC[r][m]$ at line 3 or line 3 of round r. According to the validity properties of k-simultaneous consensus and adopt-commit objects and to the structure of the lines 2 to 4, it follows that a process proposed op to $kSC[r][m]$ before τ.

There are two ways for a process to propose op to $kSC[r][m]$: either (a) it adopted it at line 25 of round $r-1$ (if $r > 1$) or (b) the process is p_i, $op \in my\_list_i[m]$, and p_i wrote op into $oper_i$ at line 24 or line 27 of round $r-1$ (if $r > 1$), or during initialization (if $r = 1$). With the same reasoning as in the previous paragraph, case (a) implies that a process proposed op to $kSC[r-1][m]$ before τ. This can be explained by case (a) at round $r-2$ only if $r > 2$, or by case (b) at round $r-2$. By iterating this reasoning, in the worst case until reaching round 1, it comes that in any case (b) happened, and that p_i necessarily proposed op to one of the $kSC[-][m]$ objects before τ. Consequently, no process p_j has op in $\ell\_hist_j[m]$ before p_i proposed it to one of the $kSC[-][m]$ objects, thus the linearization point of op is after p_i has made public the operation op.

On the other hand, if it terminates, the operation op issued by p_i ends at lines 10 or 31 after p_i computed an output for op. It can do it only at lines 9 or 20, and, in both cases, thanks to line 8 or lines 18-19, this happens only when op appears in $\ell\_hist_i[m]$. This implies that p_i either obtained a history containing op at line 5 of the same round, or writes a history containing op in $GSTATE[i][m]$ at line 30 of the same round before executing line 31, which proves that the linearization point of op is before op terminates at p_i (if it ever terminates).

Finally, according to Lemma 13, all the processes construct the same history of operations on m. Since the results locally returned are appropriately computed with compute_output() on the right prefix of the local history of m, the sequential specification of the object m is satisfied. This concludes the fact that there is a linearization of the sequence of operations applied on any object m. As any object m is linearizable, and as linearizability is a local property [46], it follows that the execution is linearizable, which ends the proof of the lemma. $\square_{Lemma\ 14}$

Lemma 15 *$\forall\ r \geq 1$, there is a process p_i such that at least one operation op output by $kSC[r]$.propose() at p_i (line 2) is such that the invocation of $AC[r][-]$.propose() by p_i returns $(commit, op)$ (line 3 or 4).*

Proof The proof is based on an observation presented in [34]. Let us first notice that, after it has received a pair (ksc_obj_1, ksc_op_1) from $kSC[r]$.propose() at line 2, a process p_{i1} invokes first $AC[r][ksc\_obj_1]$.propose(ksc_op_1) at line 3 before invoking $AC[r][ksc\_obj]$.propose$(-)$ at line 4 for any object $ksc\_obj \neq ksc\_obj_1$. If the invocation $AC[r][ksc\_obj_1]$.propose(ksc_op_1) issued by p_{i1} returns the pair $(commit, -)$, the lemma follows.

Hence, let us assume that the invocation by p_{i1} of $AC[r][ksc\_obj_1]$.propose(ksc_op_1) at line 3 returns the pair $(adopt, -)$. It follows from the "non-conflicting values" property of the adopt-commit object $AC[r][ksc\_obj_1]$, that a process p_{i2} has necessarily invoked the operation $AC[r][ksc\_obj_1]$.propose(op'), with $op' \neq ksc\_op_1$, and this invocation was issued at line 4 (if both p_{i1} and p_{i2} had invoked $AC[r][ksc\_obj_1]$.propose() at line 3, they would have obtained the same pair from the object $kSC[r]$ at line 2, and consequently, p_{i2} could not prevent p_{i1} from obtaining $(commit, -)$ from the adopt-commit object $AC[r][ksc\_obj_1]$). It follows that p_{i2} starts line 4 before p_{i1} terminates line 3. The invocation by p_{i2} of $AC[r][-]$ at line 3 involved some object ksc_obj_2 obtained by p_{i2} from its invocation of $kSC[r]$.propose() at line 2 (as seen previously, we necessarily have $ksc\_obj_2 \neq ksc\_obj_1$).

If the invocation by p_{i2} of $AC[r][ksc\_obj_2]$.propose() returns $(commit, -)$, the lemma follows. Otherwise, due to the "non-conflicting values" property of adopt-commit, there is a process p_{i3} that prevented p_{i2} from obtaining $(commit, -)$ from its invocation of the operation $AC[r][ksc\_obj_2]$.propose() at line 3. Let us notice that $p_{i3} \neq p_{i1}$ (this follows from the observation that p_{i3} started line 4 before p_{i2} terminates line 3, which itself started line 4 before p_{i1} terminates line 3, hence p_{i3} started line 4 before p_{i1} terminates line 3). The execution pattern between p_{i2} and p_{i3} is then the same as the previous pattern between p_{i1} and p_{i2}. While this pattern can be reproduced between p_{i3} and another process p_{i4}, then between p_{i4} and p_{i5}, etc., its number of occurrences is necessarily bounded because the number of processes is bounded. It then follows that there is a process p_{ix} that obtains the pair $(commit, -)$ when it invokes $AC[r][ksc\_obj_{ix}]$.propose() at line 3 (where ksc_obj_{ix} is the object returned to p_{ix} by its invocation $kSC[r]$.propose() at line 2). $\square_{Lemma\ 15}$

Lemma 16 *There is at least one object on which an infinite number of operations are executed.*

Proof This lemma follows from (a) the fact that an operation committed during some round at some process is eventually made globally visible in $GSTATE$ (lines 17, 18, and 30), (b) Lemma 15 (at every round an operation is committed at some process), and (c) the fact that the number of objects is bounded. $\square_{Lemma\ 16}$

It follows from the previous lemma, and the fact that there is a bounded number of processes, that at least one process executes an infinite number of its operations on an object. Hence the following corollary.

Corollary 4 *The algorithm is non-blocking.*

Theorem 16 *The algorithm of Figure 4.2 is a non-blocking linearizable $(k, 1)$-universal construction.*

Proof The proof follows from the previous lemmas and corollary. $\square_{Theorem\ 16}$

Generosity wrt obstruction-freedom We observe that the construction of Figure 4.2 is also obstruction-free (k, k)-universal. That is, the construction guarantees that each process will be able to complete all its pending operations in a finite number of steps, if all the other processes "hold still" long enough. Thus, if once in a while all the processes except one "hold still" long enough, then all the k objects (and not "at least one") are guaranteed to always make progress.

4.4.4 Eliminating Full Object Histories

For each process p_i and object m, the universal construction uses a shared register $GSTATE[i][m]$ to remember the sequence of all the operations that have been successfully applied to object m, as currently known to p_i. We have chosen this implementation mainly due to its simplicity. While it is space inefficient, it can be improved as follows.

- Recall that we have assumed that all the operations are unique. This can be easily implemented locally, where each process attaches a unique (local) sequence number plus its id to each operation. The (local) sequence number attached can be the number of operations the process has invoked on the object so far. Now, instead of remembering (by each process) for each object m its full history, it is sufficient that each process p_i computes and remembers only the last state of m, denoted $\ell\_state_i[m]$, plus the sequence number of the last operation successfully applied to m by each process.

- As far as the function compute_output(op, h) used at line 9 and line 20 is concerned, we have the following, where $OUTPUT[1..n]$ is an array made up of one atomic register per process. Immediately after line 18, a process p_i executes the following statements, which replace lines 19-23.

  $output_i[m] \leftarrow$ compute_output$(ac_op_i[m], \ell_state_i[m])$;
  let p_j **be** the process that invoked $ac\_op_i[m]$;
  if $(i = j)$ **then** lines 21-22
  else $OUTPUT[j] \leftarrow output_i[m]$
  end if.

  Finally, when executed by a process p_j, the line 9 is replaced by the instruction $output_j[m] \leftarrow OUTPUT[j]$.

It is easy to see that these statements implement a simple helping mechanism that allow processes, which invoke append() at line 18, to pre-compute the operation results for the processes that should invoke compute_output(op, h) at line 9. Consequently, the distributed universal construction can be easily modified to use this more space efficient representation instead of the "full history" representation.

4.5 A Contention-Aware Wait-free (k, ℓ)-Universal Construction

4.5.1 A Contention-aware non-blocking k-universal construction

Contention-aware universal construction A *contention-aware* universal construction (or object) is a construction (object) in which the overhead introduced by synchronization primitives which are different from atomic read/write registers (like k-SC objects) is eliminated in executions when there is no contention. When a process invokes an operation on a contention-aware universal construction (object), it must be able to complete its operation by accessing only read/write registers in the absence of contention. Using other synchronization primitives is permitted only when there is contention. (This notion is close but different from the notion of *contention-sensitiveness* introduced in [93].)

A contention-aware non-blocking $(k, 1)$-universal construction A contention-aware $(k, 1)$-universal construction is presented in Figure 4.3. At each round r, it uses two adopt-commit objects per constructed object m, namely $AC[2r_i - 1][m]$ and $AC[2r_i][m]$, instead of a single

one. When considering the basic construction of Figure 4.2, the new lines are prefixed by N, while modified lines are postfixed by M.

```
for each m ∈ {1,...,k} do
      my_op_i[m] ← my_list_i[m].next(); oper_i[m] ← my_op_i[m] end for.

repeat forever
(1)   r_i ← r_i + 1;
(N1)  for each m ∈ {1,...,k} do
          (tag_i[m], ac_op_i[m]) ← AC[2r_i − 1][m].propose(oper_i[m]) end for;
(N2)  if (∃ m ∈ {1,...,k} : tag_i[m] = adopt) then
(2M)      (ksc_obj, ksc_op) ← kSC[r_i].propose(ac_op_i[1..k]);
(3M)      (tag_i[ksc_obj], ac_op_i[ksc_obj]) ← AC[2r_i][ksc_obj].propose(ksc_op);
(4M)      for each m ∈ {1,...,k} \ {ksc_obj} do
              (tag_i[m], ac_op_i[m]) ← AC[2r_i][m].propose(ac_op_i[m]) end for
(N3)  end if;
lines 5- 31 of the construction of Figure 4.2
end repeat.
```

Figure 4.3: Contention-aware Non-Blocking $(k, 1)$-Universal Construction (code for p_i)

A process p_i first invokes, for each object m, the adopt-commit object $AC[2r_i-1][m]$ to which it proposes $oper_i[m]$ (new line N1). Its behavior depends then on the number of objects for which it has received the tag $commit$. If it has obtained the tag $commit$ for all the objects m (the test of the new line N2 is then false), p_i proceeds directly to the code defined by the lines 5- 31 of the basic construction described in Figure 4.2, thereby skipping the invocation of the synchronization object $kSC[r]$ associated with round r.

Otherwise, the test of the new line N2 is true and there is at least one object for which p_i has received the tag $adopt$. This means that there is contention. In this case, the behavior of p_i is similar to the lines 2-4 of the basic algorithm where, at lines 2 and 4, the input parameter $oper_i[m]$ is replaced by the value of $ac\_op_i[m]$ obtained at line N1 (the corresponding lines are denoted 2M and 4M). Moreover, at line 3, r_i is replaced by $2r_i$ (new line 3M). It is possible to reduce the number of uses of underlying k-SC synchronization objects. Such an improvement is described in Subsection 4.5.2.

Interestingly, for the case of $k = 1$, the above universal construction is the first known $contention\text{-}aware$ $(1, 1)$-universal construction.

Theorem 17 *The algorithm presented in Figure 4.3 is a non-blocking contention-aware $(k, 1)$-universal construction.*

Proof The proof first shows that the modified code provides the same safety guarantees than the previous construction. Namely, for any m, if a process p_i terminates line N3 with $tag_i[m] = commit$, then any process p_j executing line N3 ends it with $ac\_op_j[m] = ac\_op_i[m]$. Let us remark that if p_i retrieves the pair $(commit, ac\_op_i[m])$ from $AC[2r_i - 1][m]$ at line N1, it follows from the property of the adopt-commit object that any other process p_j executing this line finishes it with $ac\_op_j[m] = ac\_op_i[m]$. Consequently all processes executing lines 2M to 4M propose only this value to the k-simultaneous consensus object at line 2M or to the $AC[2r_i][m]$ object at line 4M. Moreover according to the validity of the k-simultaneous consensus object, if a process retrieves a pair (m, ksc_op) from the k-simultaneous consensus of line 2M then $ksc\_op = ac\_op_i[m]$, thus $ac\_op_i[m]$ is the only value that can be proposed to $AC[2r_i][m]$ at line 3. It follows that if a process retrieves a pair $(commit, op)$ from $AC[2r_i - 1][m]$ then any process p_j that executes lines 2M to 4M finishes line 4M with $ac\_op_j[m] = op$, while, thanks to the agreement property of $AC[2r_i - 1][m]$, any process p_h that do not execute lines 2M to 4M

also ends line N3 with $ac\_op_h[m] = op$. Additionally, if a process obtains a pair $(commit, op)$ from $AC[2r_i][m]$ while all processes obtain $(adopt, -)$ from $AC[2r_i - 1][m]$, then each process p_j executes lines 2M to 4M and thus, according to the agreement property of $AC[2r_i][m]$, obtains a pair $(-, op)$ from it and finishes line 4M with $ac\_op_j[m] = op$.

Moreover, the progress property verified by the previous construction is preserved: for any m, if a process p_i which starts line N1 with $oper_i[m] = op$, finishes the execution of line N3 before any process p_j with $oper_j[m] \neq op$ executes line N1, then p_i ends line N3 with $tag_i[m] = commit$ and $ac\_op_i[m] = op$. This comes directly from the validity properties of the k-simultaneous consensus and adopt-commit objects.

Finally, if a process executes alone, the k-simultaneous consensus object is not used and all the objects progress, while, in case of contention, as before, at least one object progresses (the first part comes from the validity property of the $AC[2r_i - 1][-]$ objects and the condition stated at line N2; the second part comes from Lemma 15).

Thanks to the previous observations, the lemmas of Theorem 16 hold with the modified code, which ends this proof. $\square_{Theorem\ 17}$

4.5.2 Contention Awareness: Reducing the Number of Uses of k-SC Objects

It is possible to reduce the number of uses of the underlying k-SC synchronization objects. This is obtained by replacing the lines N1 to N3 in Figure 4.3 as described in Figure 4.4. There is one modified line (N2M) and three new lines (NN1, NN2, and NN3).

```
(N1)   for each m ∈ {1,...,k} do
           (tag_i[m], ac_op_i[m]) ← AC[2r_i − 1][m].propose(oper_i[m]) end for;
(N2M)  if (∀ m ∈ {1,...,k} : tag_i[m] = adopt)          % ∀ m replaces ∃ m%
(2M)       then (ksc_obj, ksc_op) ← kSC[r_i].propose(ac_op_i[1..k]);
(3)            (tag_i[ksc_obj], ac_op_i[ksc_obj]) ← AC[2r_i][ksc_obj].propose(ksc_op);
(4M)           for each m ∈ {1,...,k} \ {ksc_obj} do
                   (tag_i[m], ac_op_i[m]) ← AC[2r_i][m].propose(ac_op_i[m]) end for
(NN1)      else for each m ∈ {1,...,k} do
(NN2)               if (tag_i[m] = adopt) then
                        (tag_i[m], ac_op_i[m]) ← AC[2r_i][m].propose(ac_op_i[m]) end if
(NN3)          end for
(N3)   end if.
```

Figure 4.4: Efficient Contention-aware Non-Blocking $(k, 1)$-Universal Construction (code for p_i)

More precisely, if after it has used the adopt-commit objects $AC[2r_i - 1][m]$, for each constructed object m, p_i has received only tags $adopt$ (modified line N2M), it executes the lines 2M, 3, and 4M, as in basic contention aware construction of Figure 4.3. Differently, if it has received the tag $commit$ for at least one constructed object, it invokes $AC[2r][m]$ for all the objects m for which it has received the tag $adopt$ (new lines NN1-NN3).

4.5.3 On the process side: from non-blocking to wait-freedom

The aim here is to ensure that each correct process executes an infinite number of operations on each object that progresses forever. As far as the progress of objects is concerned, it is important to notice that, while Lemma 16 shows that there is always at least one object that progresses forever, it is possible that, in a given execution, several objects progress forever.

Going from non-blocking to wait-freedom requires to add a helping mechanism to the basic non-blocking construction. To that end, the following array of atomic registers is introduced.

- $LAST\_OP[1..n, 1..m]$: matrix of atomic single-writer/multi-readers registers such that $LAST\_OP[i, m]$ contains the last operation of my_list_i invoked by p_i. Initialized to $\perp$, such a register is updated each time p_i invokes my_list_i.next() (initialization, line 11 and line 22). So, we assume that $LAST\_OP[i, m]$ is implicitly updated by p_i when it invokes the function next().

Then, for each object m, the lines 24 and 27 where is defined $oper_i[m]$ (namely, the proposal for the constructed object m submitted by p_i to the next k-SC object) are replaced by the following lines ($|s|$ denotes the size of the sequence s).

(L1) $j \leftarrow |\ell\_hist_i[m]| \bmod n + 1$; $next\_prop\_m \leftarrow LAST\_OP[j, m]$;
(L2) **if** $next\_prop\_m \notin (\{\perp\} \cup \ell\_hist_i[m])$
(L3) **then** $oper_i[m] \leftarrow next\_prop\_m$
(L4) **else** $oper_i[m] \leftarrow my\_op_i[m]$
(L5) **end if**.

This helping mechanism is close to the one proposed in [38]. It uses, for each object m, a simple round-robin technique on the process identities, computed from the current state of m as known by p_i, i.e., from $\ell\_hist_i[m]$. More precisely, the helping mechanism uses the number of operations applied so far to m (to p_i's knowledge) in order to help the process p_j such that $j = |\ell\_hist_i[m]| \bmod n + 1$ (line L1). To that end, p_i proposes the last operation issued by p_j on m (line L3) if (a) there is such an operation, and (b) this operation has not yet been appended to its local history of m (predicate of line L2). This operation has been registered in $LAST\_OP[j, m]$ when p_j executed its last invocation of $my\_list_j[m]$.next(). If the predicate of line L2 is not satisfied, p_i proceeds as in the basic algorithm (line L4).

Theorem 18 *When replacing the lines* 24 *and* 27 *by lines* L1-L5*, the algorithms of Figure* 4.2 *and Figure* 4.3 *are wait-free linearizable* $(k, 1)$*-universal constructions.*

Let us remark that requiring wait-freedom only for a subset of correct processes, or only for a subset of objects that progress forever is not interesting, as wait-freedom for both (a) all correct processes, and (b) all the objects that progress forever, does not require additional computing power.

Proof Let us first observe that the lines 1-N3 of Figure 4.3 do not access the local variables $my\_op_i[m]$, and consequently have no impact on the lines 24 and 27 replaced by the new lines L1-L5.

An increase of a local history $\ell\_hist_i[m]$ is *direct* if it occurs at line 18, and *indirect* if it occurs at line 7. Let us observe that a direct increase adds one operation to a local history. Moreover, all increases are caused by direct increases, which can then be propagated by indirect increases.

All the time instants considered in this proof are time instants after which all faulty processes have crashed. Let m be an object which progresses forever. Let p_j be a correct process such that the last operation it has written in $LAST\_OP[j, m]$ is never executed. Let $op(j, m)$ denotes this operation. The proof is by contradiction.

Let r be a round such that (a) $op(j, m)$ has been written in $LAST\_OP[j, m]$, and (b) there is a direct increase such that there is a process p_i such that $|\ell\_hist_i[m]| \bmod n + 1 = j$. Let us observe that, as the object m progresses forever and all increases are due to direct increases, both such a round r and process p_i do exist. Moreover, as it is a direct increase, p_i executed line 18 from which it follows that it executes line 24 of round r. Hence, p_i executes the new code L1-L5 of the lines 24 and we necessarily have $oper_i[m] = op(j, m)$.

If, during round r, all processes execute the new code L1-L5 of lines 24 or 27, they all starts the next round $r + 1$ with $oper_i[m] = op(j, m)$, and consequently $op(j, m)$ will be committed during

round $r+1$. In this case, $op(j,m)$ will be executed, contradicting the initial assumption. Hence, let us assume that a process p_h executes line 25 during round r. We have $oper_h[m] = ac\_op_h[m]$, where $ac\_op_h[m] = op$ is the operation committed by p_i at round r. Let us observe that we have then necessarily $|\ell\_hist_h[m]| = |\ell\_hist_i[m]| - 1$ (p_i has added op to $\ell\_hist_i[m]$ while p_h has not yet done it). We consider two cases.

- Process p_h terminates line N3 before p_i (or any other process which behaves as p_i) starts line N1. In this case, p_h terminates line N3 with the pair $(tag_i[m], ac\_op_i[m])$ equal to $(commit, op)$, and consequently adds op to $\ell\_hist_h[m]$. We have now $|\ell\_hist_h[m]| = |\ell\_hist_i[m]|$, and all the processes p_x that proceed to the round $r+2$, all verify $oper_x[m] = op(j,m)$. It follows that $op(j,m)$ will be committed during round $r+2$, which contradicts our assumption.

- Process p_i (or a process that, during round r, behaves as p_i, i.e., which has committed an operation on m –necessarily op–) starts line N1 before p_h (or a process which behaves as p_h) has terminated line N3. It follows that p_h terminates line N3 with either $ac\_op_h[m] = op(j,m)$ or $ac\_op_h[m] = op$ (the operation stored in $oper_h[m]$ and committed by p_i at round r).

  In this case, p_i has made public $\ell\_hist_i[m]$ (line 30) before p_h reads $GSTATE[i][m]$ (line 5). Hence, p_h reads the local history $\ell\_hist_i[m]$, and consequently $\ell\_hist_h[m]$ contains $\ell\_hist_i[m]$. Moreover, we also have $op \in \ell\_hist_h[m]$ when p_i executes the body of the loop of line 15 for object m. We consider two sub-cases.

  - $\ell\_hist_h[m] = \ell\_hist_i[m]$.
    - If $ac\_op_h[m] = op$: then $ac\_op_h[m] \in \ell\_hist_h[m]$, and p_h executes the new code L1-L5 of line 27. As $\ell\_hist_h[m] = \ell\_hist_i[m]$, we consequently have $oper_h[m] = op(j,m)$, from which it follows that every process p_x start the next round $r+2$ with $oper_x[m] = op(j,m)$; $op(j,m)$ is then committed during the next round, which contradicts our assumption.
    - If $ac\_op_h[m] = op(j,m)$ and the associated tag is $adopt$: p_h executes line 25, and we have $oper_h[m] = op(j,m)$. If $ac\_op_h[m] = op(j,m)$ and the associated tag is $commit$: the processes commit $op(j,m)$. In both case, $op(j,m)$is committed (at the current round or the next one), which contradicts the initial assumption.
  - $\ell\_hist_i[m]$ is a strict prefix of $\ell\_hist_h[m]$. In this case, p_h does not participate in the commitment of the operation on m that follows op in $\ell\_hist_h[m]$. It perceived it from an indirect increase of $\ell\_hist_h[m]$.

If follows from the previous reasoning that the initial assumption (namely, $op(j,m)$ is never committed) is contradicted. Consequently $op(j,m)$ is committed. As this is true for any correct process p_j and any object m that progresses forever, it follows that any correct process executes an infinite number of operations on any object that progresses forever. $\square_{Theorem\ 18}$

4.5.4 On the object side: from one to ℓ objects that always progress

Definition: (k,ℓ)-Simultaneous consensus Let (k,ℓ)-simultaneous consensus (in short (k,ℓ)-SC), $1 \leq \ell \leq k$, be a strengthened form of k-simultaneous consensus where (instead of a single pair) a process decides on ℓ pairs $(x_1, v_1), ..., (x_\ell, v_\ell)$ (all different in their first component). The agreement property is the same as for a k-SC object, namely, if (x,v) and (x,v') are pairs decided by two processes, then $v = v'$.

Notations Let (k, ℓ)-UC be any algorithm implementing the k-universal construction where at least ℓ objects always progress[2]. Let $\mathcal{ARW}[(k, \ell)\text{-}SC]$ be $\mathcal{ARW}$ enriched with (k, ℓ)-SC objects, and $\mathcal{ARW}[(k, \ell)\text{-}UC]$ be $\mathcal{ARW}$ enriched with a (k, ℓ)-UC algorithm.

A contention-aware wait-free (k, ℓ)-universal construction One can implement a contention-aware wait-free (k, ℓ)-UC algorithm on top of $\mathcal{ARW}[(k, \ell)\text{-}SC]$ as follows. This algorithm is the algorithm of Figure 4.3, where lines 24 and 27 are replaced by the lines L1-L5 introduced in Section 4.5.3, and where the lines 2M, 3M, and 4M, are modified as follows (no other line is added, suppressed, or modified).

- Line 2M: the k-simultaneous consensus objects are replaced by (k, ℓ)-simultaneous consensus objects, Hence, the result returned to a process is now a set of ℓ pairs whose first components are all distinct. It is denoted $\{(ksc\_obj_1, ksc\_op_1), ..., (ksc\_obj_\ell, ksc\_op_\ell)\}$. Let L be the corresponding set of ℓ different objects, i.e., $L = \{ksc\_obj_1, ..., ksc\_obj_\ell\}$. As already indicated, two different processes can be returned different sets of ℓ pairs.
- Line 3M: process p_i executes this line for each object $m \in L$. These ℓ invocations of the adopt-commit object (i.e., $AC[2r_i][ksc\_obj_x].\mathsf{propose}(ksc\_op_x)$, $1 \leq x \leq \ell$) can be executed in parallel, which means in any order. Let us notice that if several processes invokes $AC[2r_i][ksc\_obj_x].\mathsf{propose}()$ on the same object ksc_obj_x, they invoke it with the same operation ksc_op_x.
- Line 4M: $AC[2r_i][m].\mathsf{propose}(oper_i[m])$ is invoked only for the remaining objects, i.e., the objects m such that $m \in \{1, ..., k\} \setminus L$. As in the algorithm of Figure 4.3, the important point is that a process invokes $AC[2r_i][ksc\_obj_x].\mathsf{propose}()$ first on the set L of the objects output by the (k, ℓ)-SC object associated with the current round, and only after invoking it on the other objects.

Theorem 19 *With respect to the model $\mathcal{ARW}$, (k, ℓ)-UC and (k, ℓ)-SC have the same computational power:* (a) *a (k, ℓ)-UC algorithm can be wait-free implemented in the model $\mathcal{ARW}[(k, \ell)\text{-}SC]$, and, reciprocally,* (b) *a (k, ℓ)-SC object can be wait-free built in the model $\mathcal{ARW}[(k, \ell)\text{-}UC]$.*

This theorem shows that (k, ℓ)-SC objects are both necessary and sufficient to ensure that at least ℓ objects always progress in a set of k objects. Let us remark that this is independent from the fact that the implementation of the k-universal construction is non-blocking or wait-free (going from non-blocking to wait-freedom requires the addition of a helping mechanism, but does not require additional computational power).

Proof Proof of (a). The proof that a (k, ℓ)-UC algorithm can be implemented in the model $\mathcal{ARW}_{n,n-1}[(k, \ell)\text{-}SC]$ amounts to show that (k, ℓ)-SC allows at least ℓ objects to progress forever. If during a given round one of the processes does not verify the condition of line N2, as noticed in the proof of Theorem 17, all the objects progress. If all the processes execute lines 2M to 4M, then the reasoning of Lemma 15 holds and at least one process obtains only *commit* tags at line 3 from the ℓ adopt-commit objects associated with the ℓ objects for which it obtained operations from the (k, ℓ)-SC object associated with the corresponding round. Consequently, during any round, at least ℓ objects progress.

Proof of (b). To prove that a (k, ℓ)-SC object can be built in $\mathcal{ARW}_{n,n-1}[(k, \ell)\text{-}UC]$, let us consider an algorithm (k, ℓ)-UC where the k concurrent objects it is instantiated with are atomic

[2]It is possible to express (k, ℓ)-UC as an object accessed by appropriate operations. This is not done here because such an object formulation would be complicated without providing us with more insight on the question we are interested in.

read/write registers. Moreover, on each object m, a process p_i issues a write operation followed by read operations. When a process p_i wants to propose to the (k, ℓ)-SC object the vector $[v_i^1, ..., v_i^k]$, it invokes for each $m \in \{1, ..., k\}$, the operation $write(v_i^m)$ on the corresponding object m. Due to the (k, ℓ)-UC algorithm, each process sees at least ℓ objects progress. As soon as a process p_i sees that ℓ objects have progressed, it returns an output vector of size k containing the ℓ values written in these objects, and $\perp$ at each of the $k - \ell$ remaining entries. Hence, a process p_i returns a vector of size k with exactly ℓ non-$\perp$ entries. Moreover, it follows from the (k, ℓ)-UC algorithm that, the processes see the same sequence of operations on each object. Hence, if p_i returns $v \neq \perp$ and p_j returns $v' \neq \perp$ for the same entry m of their output arrays, these values have been written by the same write operation, and are consequently such that $v = v'$, which concludes the proof. $\square_{Theorem\ 19}$

4.6 Conclusion and Remarks on the (k, l)-Universal Constructions

In this chapter we presented a new modular approach to build (k, l)-universal constructions. The proposed constructions can guarantee the progress of more than one of the simulated object if provided with stronger simultaneous consensus objects, and these objects have been shown necessary to achieve this goal. We described possible improvements of our universal construction to reduce its memory footprint, to avoid the use of the simultaneous consensus objects in absence of contention and to ensure that the correct processes can issue an infinite number of operations on the objects that progress forever.

We hope that this work will help to understand better how the k-set agreement allows the implementation of shared objects and what are the inherent limitations of such constructions. A possible next step for this work would be to reuse parts of it in order to build a distributed simulation of k processes communicating through a shared memory, some of these simulated processes being prone to crashes. It would bring interesting reductions in terms of computability, of the same kind as those deduced from the BG-simulation [15].

Chapter 5

Computing in the Presence of Concurrent Solo Executions

This chapter presents a new hierarchy of wait-free models that spans from $\mathcal{AMP}$ to $\mathcal{ARW}$. These models are built as a variations of the $\mathcal{IIS}$ model, and have the property of allowing several processes to execute in isolation. Thanks to their round-based executions using one-shot communication objects (similarly to $\mathcal{IIS}$), a topological study of the set of allowed executions and of the tasks that can be solved in these models is possible and shows that the obtained hierarchy is strict in terms of computability. These results have been published in [43].

Section 5.1 describes the context of this work, some results related to the topological study of asynchronous distributed systems and how our results relate to the state of the art. Section 5.2 then formally defines the proposed d-solo models and the communication objects they use. Section 5.3 introduces the notions of colorless tasks and algorithm and describes the topological representation of the possible executions of the colorless algorithm in the d-solo models. Section 5.4 characterizes the tasks that can be computed in one of these models. Section 5.5 introduces a new problem, named (d, ϵ)-solo approximate agreement and shows that, in the general case, it can be solved in the d-solo model but not in the $(d+1)$-solo model. Section 5.6 discusses some properties of the (d, ϵ)-solo approximate agreement problem and how it relates to the k-set agreement problem. Finally Section 5.7 concludes the chapter.

5.1 Context and Introduction

Distributed computability When looking at the communication medium and assuming asynchronous processes prone to crash failures, a read/write system and a message-passing system have the same computability power if and only if less than half of the processes may crash [8]. If a majority of the processes may crash, the message passing model is weaker than the shared memory model because partitions can occur.

The power of a distributed model has been studied in detail with respect to *tasks*, which are the distributed equivalent of functions in sequential computing. Each process gets only one part of the input, and after communicating with the others, decides on an output value, such that collectively, the various local outputs produced by the processes respect the task specification, which is defined from the local inputs of the processes. This work concentrates on the class of *colorless tasks* (e.g., [13, 41, 42]), where the specification is in terms of possible inputs and outputs, but without referring to which process gets which input or produces which output. Among the previously studied notable tasks, many are colorless, such as consensus [31], set agreement [22], approximate agreement [27], and loop agreement [41], while some are not, like renaming [9].

Wait-freedom and solo execution In a wait-free model where processes must satisfy the wait-freedom liveness condition, a process has to make progress even in the extreme cases where all other processes have crashed, or are too slow, and consequently be forced to decide without knowing their input values. Hence, for each process, there are executions where this process perceives itself as being the only process participating in the computation.

More generally, we say that a process executes *solo* if it computes its local output without knowing the input values of the other processes.

Two extreme wait-free models: shared memory and message passing In a model where processes communicate by reading and writing shared registers, at most one process can run solo in any execution. This is because, when a process runs solo, it writes and reads from the shared memory, and eventually writes its decision. Any other process that starts running, will be able to read the history left by the solo process in the memory.

When considering message-passing communication, all processes may have to run solo concurrently in the extreme case, where messages are arbitrarily delayed, and each process perceives the other processes as having crashed. Only tasks that can be solved without communication can be computed in this model.

Investigating the computability power of intermediary models The aim this chapter is to study the computability power of asynchronous models in which several processes may run solo in the same execution. More precisely, assuming that up to d processes may run solo, we try to address the following questions:

- How to define a computation model in which up to d processes may run solo?
- Which tasks can be computed in such a model?

The aim is to study these questions in a clean theoretical framework, and to investigate models weaker than the basic wait-free read/write model. However, we hope that our results are relevant to other intermediate models, such as distributed models over fixed or wireless networks.

To simplify the technical development, following [14], we propose a theoretical round-based framework, similar to $\mathcal{IIS}$. In the proposed model, the processes exchange information through communication object whose behavior is close from the one of immediate snapshots objects. This allows us to study the executions of this model using combinatorial topology as in [45].

Contributions The following contributions answer the previous questions:

- The definition of a family of $d\text{-}solo\ models$, each parametrized with an integer d, $1 \leq d \leq n$. The 1-solo model corresponds to the $\mathcal{IIS}$ model (which is equivalent to the read/write wait-free model [14]), while the n-solo model corresponds to the round-based wait-free message-passing model.
- A characterization of the set of colorless tasks that can be solved in the d-solo model, $1 \leq d \leq n$. Via a new form of complex subdivisions, this characterization connects topology with $colorless$ algorithms.
- Any d-*solo* model with $d \geq 2$, is weaker than the read/write wait-free model, yet there are natural, non-trivial tasks that can be solved in the d-*solo* model. One of these tasks, called $(d, \epsilon)\text{-}solo\ approximate\ agreement$ (in short (d, ϵ)-SAA) is such that (d, ϵ)-SAA can be solved in the d-solo model, for any $\epsilon > 0$, but not in the $(d+1)$-solo model. Hence, more tasks can be solved in the d-solo model than in the $(d+1)$-solo model, for $1 \leq d < n$, which establishes a hierarchy of solo models.

- Finally, the d-solo model is related to d-set agreement. This relation shows that, for $d < n$, d-set agreement is strong enough to solve (d, ϵ)-solo approximate agreement but is too weak to solve $(d-1, \epsilon)$-solo approximate agreement in the wait-free message-passing model. This provides us with a better insight on a bound on the "maximal partitioning" allowed to solve $(d-1, \epsilon)$-solo approximate agreement in the wait-free message-passing model.

The (d, ϵ)-solo approximate agreement task is a generalization of approximate agreement [27]. The input of each process consists of a point in the Euclidean space $\mathbb{R}^N$ ($N \geq d$). The $validity$ property states that each process p_i has to decide a point which is in the convex hull of all the input points. The $agreement$ property states that at most d processes may decide any point in the convex hull of the input points (let CH be the convex hull defined by these at most d points), while the other processes have to decide values whose distance to CH is at most ϵ. Actually, the convex hull of solo processes is an "attractor" for the set of decided values.

When $d = 1$, validity and agreement imply that the Euclidean distance between any pair of points decided by the processes has to be upper bounded by a predefined constant. Thus, $(1, \epsilon)$-solo approximate agreement problem in $\mathbb{R}^m$ is essentially the problem that has been recently considered in the context of t Byzantine failures and asynchronous message-passing systems [60, 94], where it is shown that it can be solved iff $n > t(m+2)$.

The colorless tasks that are solvable in the wait-free $\mathcal{IIS}$ model have been characterized in [42]. Due to the simulations in [14, 36], this characterization holds for the usual read/write wait-free model. Section 5.4 extends the characterization of [42] to the d-solo model, $1 \leq d \leq n$. Our characterization in terms of colorless algorithms permits the use of standard subdivisions, instead of chromatic subdivisions used in previous approaches. We believe colorless algorithms are interesting in themselves, and indeed, for $d = 1$, if a colorless task is solvable, it is solvable by a colorless algorithm. For $d > 1$ we defer the proof that colorless algorithms and general algorithms can solve a very similar class of tasks.

One of the central results of topology is the Simplicial Approximation Theorem [68], which establishes what is a "discrete version" of a continuous map. This theorem is also central for the wait-free characterization theorem of [45] and its t-resilient extension (e.g., [42]). However, this theorem cannot be used in a d-solo model, $d > 1$, because it is no longer the case that the diameter of the simplices in a subdivision is reduced. Not even the Relative Simplicial Approximation Theorem [95] can be directly used.

Finally, it is important to notice that our d-solo model addresses different issues than the d-concurrency model of [34], where it is shown that with d-set agreement any number of processes can emulate d state machines of which at least one remains highly available. While d-concurrency is used to reduce the concurrency degree to at most d processes that are always allowed to cooperate, d-solo allows up to d processes to run independently (i.e., without any cooperation).

5.2 The d-Solo Model: Communication Object and Iterated Model

Rounds and communication objects The d-solo model is defined following the same pattern as the $\mathcal{IIS}$ model: the processes execute rounds sequentially and asynchronously and communicate only through the distributed object associated to the current round. The communication object $CO[r]$ associated with each round r is the only means for the processes to communicate during round r: as in [29] the rounds are *communication-closed*.

More precisely, $CO[r]$ is a one-shot object (i.e., each process accesses it only once) which provides the processes with a single operation denoted $\mathsf{communicate}(i, v)$, where v is the value that the invoking process p_i wants to communicate to the other processes during round r. Such an invocation returns to p_i a set of pairs (process identity, value) deposited into $CO[r]$ by other

processes during round r.

Iterated model Each process p_i executes the algorithm skeleton described in Figure 5.1, where the local computation parts are related to the particular task that is solved. The local variable r_i is the local round number, ℓs_i contains p_i's local state, while $view_i$ contains all the pairs $(j, \ell s_j)$ communicated to p_i during the current round. The local transition function $\delta_i()$ defines the new local state of p_i according to its previous local state and the pairs $(j, \ell s_j)$ it has obtained from $CO^d[r]$ (the parameter d is explained below in Section 5.2.1). To solve a task, it is necessary to instantiate accordingly $\delta_i()$, the predicate decision() and the function dec_val(): decision() allows p_i to decide, while dec_val() allows it to compute the decided value. As we are interested in computability and not efficiency, we assume a *full information* algorithm, i.e., at the end of each round r_i, ℓs_i contains the value of $view_i$, and δ_i can be task independent. However, we will see in Section 5.3 that in some cases, tasks can be solved without communicating all a process knows.

```
(1)  r_i ← 0; ℓs_i ← initial local state;
(2)  loop forever r_i ← r_i + 1; view_i ← CO^d[r_i].communicate(i, ℓs_i);
(3)               ℓs_i ← δ_i(ℓs_i, view_i); if decision(ℓs_i) then dec_val(ℓs_i) end if
(4)  end loop.
```

Figure 5.1: Generic iterated model

5.2.1 Communication object

The communication objects $CO^d[1]$, $CO^d[2]$, etc., of an execution are parametrized by a solo-dimension d, $1 \leq d \leq n$. As previously indicated, an object $CO^d[r]$ contains a set of pairs, one per process. Each pair (i, v) is such that i is a process index and v the value communicated by p_i, and $CO^d[r]$ contains at most one pair per process.

Definition The behavior of every object CO^d is defined as follows. Considering an execution during which each of the n processes $\{p_1, \ldots, p_n\}$ accesses the object (at most once) using its local state ℓs_i as input, one can represent this execution by an *ordered partition*, i.e., a tuple of non-empty sets $(P_1, \ldots, P_z)$ such that (1) for any distinct $i, j \in \{1, \ldots, z\}$: $P_i \cap P_j = \emptyset$, and (2) $\bigcup_{i=1}^{z} P_i = \{p_1, \ldots, p_n\}$. From an operational view, the ordered partition $(P_1, \ldots, P_z)$ describes the sequence of concurrent accesses to the object CO^d.

The behavior of CO^d is defined from a *d-ordered partition*, where a d-ordered partition is an ordered partition $(\pi_1, \ldots, \pi_{z'})$ such that $0 \leq |\pi_1| \leq d$ (the size of the first set of the partition can be 0 and cannot exceed d). More precisely, the d-ordered partition $(\pi_1, \ldots, \pi_{z'})$ associated with CO^d is:

- If $|P_1| > d$: $(\pi_1, \ldots, \pi_{z'}) = (\emptyset, P_1, \ldots, P_z)$, and
- If $|P_1| \leq d$: $(\pi_1, \ldots, \pi_{z'}) \in \{(\emptyset, P_1, \ldots, P_z), (P_1, \ldots, P_z)\}$.

$(\pi_1, \ldots, \pi_{z'}) = (P_1, \ldots, P_z)$ captures the cases where, initially, d (or less) processes execute solo. In the other cases we have $(\pi_1, \ldots, \pi_{z'}) = (\emptyset, P_1, \ldots, P_z)$, because initially either too many processes execute concurrently (first item), or, while no more than d processes execute concurrently, none of them executes solo.

The values $view_i$, $1 \leq i \leq n$, obtained by the processes when the behavior of CO^d is represented by the *d-ordered partition* $(\pi_1, \ldots, \pi_{z'})$ are defined as follows:

$$(i \in \pi_1) \Rightarrow (view_i = \{(i, \ell s_i)\}), \quad \text{and}$$

$$(x > 1 \wedge i \in \pi_x) \Rightarrow \big(view_i = \{(j, \ell s_j) \ : \ j \in \pi_y \wedge y \leq x\}\big).$$

This means that the view of each process p_i belonging to π_1 (where $0 \leq |\pi_1| \leq d$) contains only its own contribution, namely the pair (i, ls_i). Differently, the view of a process p_i in π_x, where $x > 1$, contains all the pairs $(j, \ell s_j)$ deposited in CO^d by the processes p_j of the sets π_y such that $y \leq x$. Thus, each process of π_1 appears as executing solo, while each other process of a set p_x, $x \neq 1$, sees the contributions provided (a) by all the processes p_i belonging to the "previous" sets π_y ($y < x$), and (b) by all the processes from its "concurrency" set π_x. (The immediate snapshot object described in [13] implements CO^d for $d = 1$.)

Object properties Given an object CO^d, the next properties follow from its definition.

- Solo execution upper bound. $0 \leq |\{i \text{ such that } |view_i| = 1\}| \leq d$.
- Self-inclusion. $\forall\, i : (i, -) \in view_i$.
- Containment. $\forall\, i, j : \big((|view_i| \leq |view_j|) \wedge |view_j| > 1)\big) \Rightarrow (view_i \subseteq view_j)$.

5.2.2 Examples of communication objects

Considering a system of $n = 3$ processes, this section describes two communication objects, corresponding to the cases $d = 1$, and $d = n - 1 = 2$. (Their aim is also to show connection between these objects and topology.)

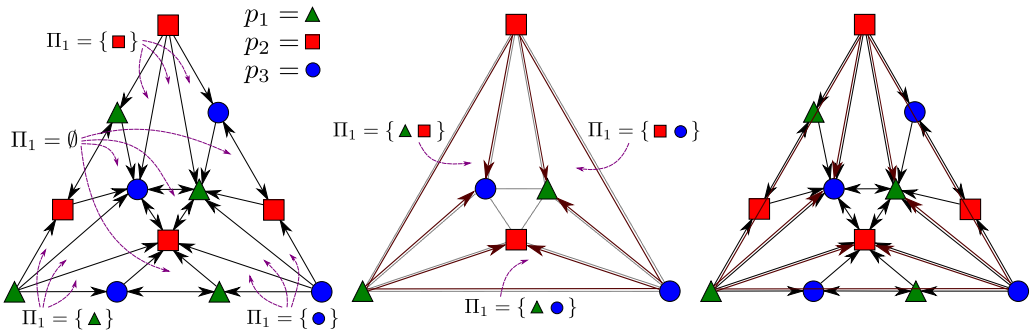


Figure 5.2: All possible executions for 3 processes

Object $CO^1[r]$ All the possible behaviors of $CO^1[r]$ that can occur are described on the left of Figure 5.2. An arrow from p_i to p_j means that the set $view_j$ (obtained by p_j when it returns from the invocation $CO^1[r].\mathsf{communicate}(j, -)$) is such that $(i, \ell s_i) \in view_j$. On the contrary, the absence of an arrow from p_i to p_j means that $(i, \ell s_i) \notin view_j$. In the topology parlance, the internal triangles are simplices defining the possible subdivision of the (external triangle which defines the) complex associated with the execution at the beginning of round r.

The possible sets π_1 that can appear during an execution of $CO^1[r]$ are indicated for each small triangle (simplex) on the figure at the left. To simplify the notation, let $v_i = \ell s_i[r-1]$. As an example, the small triangle as the center corresponds to the case where $\pi_1 = \emptyset$ and $view_1 = view_2 = view_3 = \{(1, v_1), (2, v_2), (3, v_3)\}$. For the three triangles at the bottom of the figure at the left, we have the following:

- Small triangle at the left side: $\pi_1 = \{p_1\}$, $view_1 = \{(1, v_1)\}$, $view_3 = \{(1, v_1), (3, v_3)\}$, and $view_2 = \{(1, v_1), (2, v_2), (3, v_3)\}$.

- Small triangle in the middle: $\pi_1 = \emptyset$, $view_1 = view_3 = \{(1, v_1), (3, v_3)\}$, and $view_2 = \{(1, v_1), (2, v_2), (3, v_3)\}$.
- Small triangle on the right side: $\pi_1 = \{p_3\}$, $view_3 = \{(3, v_3)\}$, $view_1 = \{(1, v_1), (3, v_3)\}$, and $view_2 = \{(1, v_1), (2, v_2), (3, v_3)\}$.

It is easy to see that the previous iterated computation model, where the communication objects are instantiated with $d = 1$, is nothing more than the $\mathcal{IIS}$ model.

Object $CO^2[r]$ The possible behaviors of $CO^2[r]$ are represented on the right side of Figure 5.2. The new behaviors added to the ones of $CO^1[r]$ are represented in the middle of Figure 5.2 (the figure at the right is consequently the "addition" to the figure at the left of the possible behaviors described in the middle).

The new additional values for π_1 are described on the figure in the middle. The case $\pi_1 = \{1, 3\}$ that appears at the bottom of the figure represents the execution in which each of p_1 and p_3 executes as if it was alone: none of them sees the pair value communicated by the other. Differently p_3 sees both of them. Hence, this triangle represents the additional execution where $view_3 = \{(3, v_3)\}$, $view_1 = \{(1, v_1)\}$, and $view_2 = \{(1, v_1), (2, v_2), (3, v_3)\}$.

It is easy to see that this model is weaker than the base wait-free asynchronous read/write model: in the execution corresponding to the bottom triangle where $\pi_1 = \{1, 3\}$, none of p_1 and p_3 "writes" its pair before the other. More generally, if $d = n$, the object $CO^n[r]$ gives an account wait-free message-passing executions where, due to message asynchrony and process crashes, it is possible that an arbitrary number of processes do not receive messages from the other processes.

5.2.3 A spectrum of solo models

It follows from their definition that CO^d is stronger (more constraining) than CO^{d+1} in the sense that the subdivided complex of CO^d is included the one of C^{d+1}. Intuitively, this means that CO^d includes "more synchrony" than CO^{d+1}.

The d-solo model The generic framework described in Figure 5.1 instantiated with CO^d objects is called the *d-solo model*. It is denoted $\mathcal{ACS}^d$($\mathcal{ACS}$ stands for Asynchronous Concurrent Solo).

Hierarchy of d-solo models Let $A \succeq_T B$ mean that any task that can be solved in the model B can be solved in the model A, and $A \simeq_T B \stackrel{def}{=} (A \succeq_T B) \wedge (B \succeq_T A)$.

Let $\mathcal{ARW}$ denote the base wait-free (asynchronous) read/write model. It follows from the fact that (for task solvability) the $\mathcal{IIS}$ model and $\mathcal{ARW}$ have the same computability power [14], and $\mathcal{IIS}$ is nothing more than $\mathcal{ACS}^1$, that we have $\mathcal{ARW} \simeq_T \mathcal{ACS}^1$.

Let $\mathcal{AMP}$ denote the classical (non-iterated) message-passing system where up to $(n-1)$ processes may crash. As all processes except one may crash and communication is asynchronous (hence messages can be arbitrarily delayed), the tasks that can be solved in $\mathcal{AMP}$ are the tasks that can be wait-free solved without communication. But, this set of tasks is exactly the set of tasks that can be solved in $\mathcal{ACS}^n$. Hence, $\mathcal{ACS}^n \simeq_T \mathcal{AMP}$.

It follows from the definition of the communication objects CO^d and CO^{d+1} that any task solvable in $\mathcal{ACS}^{d+1}$ is solvable in $\mathcal{ACS}^d$. We have consequently the following hierarchy of models:

$$\mathcal{ARW} \simeq_T \mathcal{ACS}^1 \succeq_T \ldots \succeq_T \mathcal{ACS}^d \succeq_T \ldots \succeq_T \mathcal{ACS}^n \simeq_T \mathcal{AMP}.$$

We will see in Section 5.5 that $A \succeq_T B$ can be replaced by $A \succ_T B$ (all the tasks solvable in B are solvable in A, and there is one task solvable in A and not in B).

5.3 Colorless Tasks and the d-Solo Model

This section focuses on colorless tasks that can be solved in the d-solo model. After having defined colorless tasks it shows that, for these tasks, one can use a restricted form of the algorithm in Figure 5.1. It then, introduces the notions of a (d, R)-subdivision task and a (d, R)-agreement task. Detailed definitions of the involved topology notions, can be found in [39] or [44].

5.3.1 Colorless tasks

A colorless task is a special kind of task where the processes cannot use their ids during the computation. This implies that the task specification is not in terms of ids. A colorless task specifies which sets of values are valid input configurations, and which are valid output decisions, but not which value is assigned to which process. Thus, a process may adopt the input value or the output value of another process.

Formally, a *colorless task* is a triple $(\mathcal{I}^*, \mathcal{O}^*, \Delta^*)$, where $\mathcal{I}^*$ is a colorless *input complex*, $\mathcal{O}^*$ is a colorless *output complex*, and $\Delta^* : \mathcal{I}^* \to 2^{\mathcal{O}^*}$ is a *carrier map*. A colorless complex is a family of sets, over some basic set of values, such that if a set is in the complex, then all its subsets are also in the complex. A set in the complex is called a *simplex*. Simplices of size 1, are called *vertices*, and of size 2, *edges*. Indeed, a graph is a 1-dimensional complex. In the case of a colorless complex, a vertex is just a value, either an input or an output value, while in a colored complex, a vertex is a pair of values, one is a process id, and the other is an input our output value. If σ is an input simplex in $\mathcal{I}^*$, the carrier map $\Delta^*(\sigma)$ is a subcomplex of $\mathcal{O}^*$ satisfying *monotonicity:* $\forall \sigma, \sigma' \in \mathcal{I}^* \ : \ \Delta^*(\sigma \cap \sigma') \subseteq \Delta^*(\sigma) \cap \Delta^*(\sigma')$.

Operationally, the meaning of a colorless task is the following. If $\sigma \in \mathcal{I}^*$, then the processes can start an execution with input values from σ; different processes may propose the same vertex or different vertices from σ. Processes eventually decide (not necessarily distinct) vertices that belong to the same output simplex $\tau \in \mathcal{O}^*$, such that $\tau \in \Delta^*(\sigma)$. If the system consists of n processes, then the processes can start with at most n different input values, and hence, processes will never start on a simplex σ of $\mathcal{I}$ of dimension greater than $n-1$ (the dimension of σ is $|\sigma|-1$). Thus, for n processes, only the simplices of $\mathcal{I}$ of dimension $\leq n-1$ are relevant, i.e., the $n-1$ *skeleton* of $\mathcal{I}$, denoted Skel$^{n-1}\mathcal{I}$. For example, in a system of two processes, $n = 2$, only the 1-skeleton of $\mathcal{I}$ is of interest, which is the graph consisting of the vertices and 1-simplices of $\mathcal{I}$.

5.3.2 Colorless algorithms

A *colorless algorithm* is an algorithm in the form of Figure 5.1, but where the local computation made by δ_i in line (3) is very restricted. Although a colorless algorithm is not as powerful as an algorithm with no restrictions, it simplifies that exposition, and in the full version we show that they can solve a similar class of colorless tasks.

Informally, in a colorless algorithm processes behave in an anonymous way: processes consider the shared memory as if it is a set. (A colorless complex is denoted with a $*$ superscript, as in $\mathcal{K}^*$.) In each round, a process deposits its input in the set, and gets back a view of the contents of the set. If two processes deposit the same value in the set, only one copy is stored. When a process gets back a set of values, there is no information of which process deposited which value. A process "forgets" which is its own value in the set. The set of values that a process receives at the end of a round, becomes its input to the next round.

Formally, in an execution, the initial local state of a process p_i is a vertex v_i of $\mathcal{I}^*$, and is assigned in line 1 to ℓs_i. Furthermore, the set of all initial states v_i (not necessarily distinct) is a simplex σ of $\mathcal{I}^*$. We may write, $\sigma = \{\ell s_1[0], \ldots, \ell s_n[0]\}$, where $\ell s_i[0]$ denotes the initial value of

ℓs_i. Notice that $|\sigma|$ may be less than n because different processes may start with the same input value.

The local transition δ_i eliminates process ids. Namely, during any round r and for any process p_i, if we denote by $\ell s_i[r]$ the value of ℓs_i at the end of round r, in line 2 of the algorithm, $view_i$ is assigned the value returned by $CO^d[r].\mathsf{communicate}(i, \ell s_i[r-1])$, and this value is a set of pairs $\{(i_1, \ell s_{i_1}[r-1]), \ldots, (i_k, \ell s_{i_k}[r-1])\}$ that includes ids $i_1, \ldots, i_k$, but when the function δ_i is applied to this set it returns a set $\sigma_i^r = \{\ell s_{i_1}[r-1], \ldots, \ell s_{i_k}[r-1]\}$. We assume every process executes the same number of rounds, $R \geq 0$, and in the last round, produces an output value $\mathsf{dec\_val}(\ell s_i)$ (all processes use the same function $\mathsf{dec\_val}$).

For an R round colorless algorithm in the d-dimensional model, the *algorithm complex* is defined as follows. For each input simplex $\sigma \in \mathcal{I}^*$, the subcomplex $\mathcal{P}^*(\sigma)$ represents the executions r where all processes start with inputs from σ (at least one process starts with each of the vertices in σ). Moreover, in the algorithm complex for the d-dimensional model we do not want to include the $(d-1)$-dimensional model, so we consider only runs where the processes that in a round see more than one process, they see at least $d+1$ processes. The complex $\mathcal{P}^*(\sigma)$ contains a top dimensional simplex $\tau = \{\ell s_i\}$ for each such R round execution of the algorithm starting in σ, where the vertices ℓs_i of τ are the values of $\ell s_i[r]$ at the end of this execution, for each process p_i (without repetitions, as the simplex is a set). The complex $\mathcal{P}^*$ is the union of $\mathcal{P}^*(\sigma)$ over all $\sigma \in \mathcal{I}^*$. It is easy to prove that $\mathcal{P}^*(\cdot)$ is a strict carrier map from $\mathcal{I}^*$ to the algorithm complex $\mathcal{P}^*$.

We will explain the significance of the next lemma later on, when we discuss subdivisions.

Lemma 17 *Consider a 1-round colorless algorithm and an input simplex $\sigma \in \mathcal{I}^*$. The simplices of $\mathcal{P}^*(\sigma)$ are of the form $\tau = \{\tau_1, \ldots, \tau_z\}$, where each $\tau_i \subseteq \sigma$, and there is an $l, 0 \leq l \leq d$ such that (1) for all $i, 0 \leq i \leq l$, $|\tau_i| = 1$, so $\cup_{0 \leq i \leq l} \tau_i$ is a face σ' of σ, (2) for all $j, l < j \leq z$, $\sigma' \subsetneq \tau_j$, and (3) for all $j, l < j \leq z-1$, $\tau_j \subsetneq \tau_{j+1}$.*

Proof Each simplex τ associated to a sequence of faces $F_0, \ldots, F_z$ and to an integer l as above, corresponds to an allowed behavior for the object CO^d. Namely the one represented by the d-ordered set partition $(\bigcup_{i=1}^{l} F_i, F_{l+1} \setminus \bigcup_{i=1}^{l} F_i, \ldots, F_z \setminus \bigcup_{i=1}^{z-1} F_i)$. Since $0 \leq l \leq d$, the set $\bigcup_{i=1}^{l} F_i$ contains at most d values, moreover (2) and (3) imply that the sets of the partition are pairwise disjoint

Reciprocally, if $(\pi_1, \ldots, \pi_z)$ corresponds to an allowed behavior for the object CO^d, then one can build a sequence of faces $(F_1, \ldots, F_{z+l-1})$ by choosing $l = |\{\ell s_i,$
$p_i \in \pi_1\}|$, $\{F_1, \ldots, F_l\} = \{\{\ell s_i\}, p_i \in \pi_1\}$ and $\forall i, l < i < z+l-1 \ : \ F_i = \bigcup_{j=1}^{i+|\pi_1|-1} \ell s_j$. The properties of the communication object CO^d ensure that $l \leq |\pi_1| \leq d$, but also that the properties (2) and (3) hold. Consequently the simplex τ whose vertices are the faces $F_i, 1 \leq i \leq z+l-1$, belongs to $\mathcal{P}^*(\sigma)$. $\square_{Lemma\ 17}$

If $\mathcal{P}^*(\cdot)$ is a carrier map from $\mathcal{I}^*$ to the algorithm complex $\mathcal{P}^*$, and $\mathsf{dec\_val}$ is a simplicial map from $\mathcal{P}^*$ to $\mathcal{O}^*$, we say that $\mathsf{dec\_val}$ *is carried by* Δ^* if for each $\sigma \in \mathcal{I}^*$ and each $\tau \in \mathcal{P}^*(\sigma)$, the simplex $\mathsf{dec\_val}(\tau)$ belongs to $\Delta^*(\sigma)$.

Lemma 18 *If the colorless task $(\mathcal{I}^*, \mathcal{O}^*, \Delta^*)$ is solvable by a colorless algorithm then there exists an algorithm complex $\mathcal{P}^*$, and a simplicial map $\mathsf{dec\_val}$ from $\mathcal{P}^*$ to $\mathcal{O}^*$ that is carried by Δ^*.*

Proof The output value decided by a process in line 3 is based on ℓs_i, which is a set of values, with no process ids. If the r-round colorless algorithm solves the colorless task $(\mathcal{I}^*, \mathcal{O}^*, \Delta^*)$, at the end of the r-th round, processes have to decide an output value, by executing $\mathsf{dec\_val}(\ell s_i)$ in line 3. The result of $\mathsf{dec\_val}(\ell s_i)$ is a vertex in $\mathcal{O}^*$. Different processes may decide different vertices as

long as they belong to the same simplex τ of $\mathcal{O}^*$. Moreover, if $\sigma \in \mathcal{I}^*$ is the input simplex of the execution, the output simplex τ must be in $\Delta^*(\sigma)$, to satisfy the task's specification. $\square_{Lemma\ 18}$

5.3.3 (d, R)-Subdivision and (d, R)-agreement tasks

The (d, R)-subdivision task Which is the simplest task a colorless algorithm can solve in the d-dimensional model? It is the task solved when each process executes R rounds, then stops, and its decision function is the identity! Namely, $\mathsf{dec\_val}(\ell s_i) = \ell s_i$ i.e. a process decides the set of values $\ell s_i[R]$ it retrieves from the communication object during the R^{th} round. Given any input complex $\mathcal{I}^*$ and any integer $R \geq 0$, we call this task the (d, R)-*subdivision task* over $\mathcal{I}^*$. The output complex $\mathcal{O}^*$ of this task is of course equal to the algorithm complex $\mathcal{P}^*$, with the simplicial map $\mathsf{dec\_val}$ being the identity.

For the carrier map, $\Delta^*(\sigma)$ includes all simplices τ that correspond to executions starting in σ, i.e., $\Delta^*(\sigma) = \mathcal{P}^*(\sigma)$. In particular, for $R = 0$, $\mathcal{I}^* = \mathcal{O}^*$, and Δ^* is the identity carrier map, which sends a simplex σ to the complex consisting of σ and all its faces (which we often denote by σ, abusing notation).

By definition, the (d, R)-subdivision task over $\mathcal{I}^*$ is solvable in the d-dimensional model, and moreover, by a colorless algorithm. In fact, it is the basic building block to solve every other colorless task, as shown in Theorem 20. We will justify the name "subdivision task" when we see how to specify the task without resorting to executions of some model in Section 5.3.4.

The (d, R)-agreement task When the vertices of $\mathcal{I}^*$ are points in Euclidean space, the (d, R)-subdivision task can be used directly to solve a task that we call (d, R)-*agreement task* over $\mathcal{I}^*$, which is defined combinatorially in Section 5.5. In the (d, R)-subdivision task, processes propose sets of values in each round. We can encode such a set of values as its barycenter b, and then the process can directly propose b. We shall see in Section 5.5, that, although both tasks are essentially the same, when we work with barycenters processes compute output values within ϵ of each other (except for at most d processes that may run solo), and we can make ϵ as small as we want, by choosing a large enough value of R.

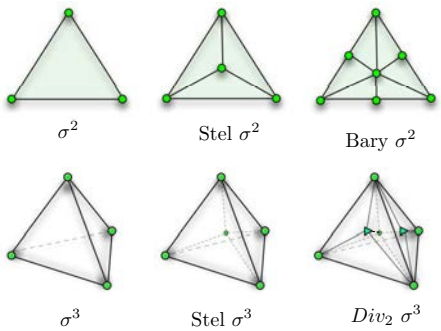
Operationally, the (d, R)-agreement task over $\mathcal{I}^*$ is defined as follows. Processes execute R rounds of a colorless algorithm in the d-dimensional model. In each round r, each process p_i computes its value $\ell s_i[r]$ that will be the input to the next round, in line 3 of the algorithm, by taking the barycenter of the values that it gets back from the object in line 2. The barycenter computed in round R is the output of of the process.

5.3.4 The structure of colorless algorithms

The structure of a colorless complex is explained in terms of *subdivisions*. Examples of subdivisions of a simplex are illustrated on the figure that follows at the right of the page.

Perhaps the simplest subdivision is the *stellar* subdivision. Given a complex (abusively denoted σ^m) consisting of an m-simplex $\sigma^m = \{s_0, \ldots, s_m\}$ and all its faces, the complex $\text{Stel}(\sigma^m, b)$ is constructed by taking a *cone* with apex b over the boundary complex $\partial\sigma^m$.

The *barycentric* subdivision, Bary σ^m, is perhaps the most widely used in topology. A simplex τ is in Bary σ^m if and only if there exists



a sequence $\sigma_0 \subset \ldots \subset \sigma_z$ of faces of σ^m, and the set of vertices of τ is the set of the barycenters of the these faces, denoted $\hat{\sigma}_i, 0 \leq i \leq z$.

For the d-solo models, we need to define a family of subdivisions that goes from the stellar to the barycentric subdivision. The *d-dimensional subdivision* of a complex $\mathcal{K}$ denoted $\text{Div}_d\ \mathcal{K}$, is the barycentric subdivision of $\mathcal{K}$ relative to $\text{Skel}^{d-1}\mathcal{K}$. Intuitively, we do not subdivide $\text{Skel}^{d-1}\mathcal{K}$ because we consider executions where up to d processes run solo, they get their own view in an invocation of a CO^d object. See the construction of Figure 5.3. As usual, the *R-iterated d-dimensional subdivision*, $\text{Div}_d^R\ \mathcal{K}$, is obtained by repeating the subdivision process R times.

```
(1)  Div_d Skel^{d-1}σ^m ← Skel^{d-1}σ^m; % each vertex is labeled by its name
(2)  for k from d to m do % Construct Div_d Skel^k σ^m %
(3)      for each simplex σ^k in σ^m do
(4)          insert a vertex b in the barycenter of σ^k;
             % this barycenter is labeled with the set of vertices of σ_k
(5)          construct the cone with apex at b over Div_d ∂σ^k;
             % over the already subdivided boundary of σ^k %
(6)          add the cone to Div_d Skel^k σ^m
(7)      end for loop
(8)  end for loop.
```

Figure 5.3: Constructing the subdivision $\text{Div}_d\ \sigma^m$ of a simplex σ^m for the d-solo model

The next lemma follows from the fact that the construction of Div_d in Figure 5.3 corresponds exactly to the description given in Lemma 17, and the fact in the system there are n processes, so they can start with at most n different input values (so only the input simplices in $\mathcal{I}^*$ of dimension at most $n-1$ are relevant).

Lemma 19 *If $\mathcal{P}_R^*$ is the R-round algorithm complex of a colorless algorithm in the d-solo model with input complex $\mathcal{I}^*$, then $\mathcal{P}_R^*$ is an R-iterated, d-dimensional subdivision of the $n-1$ skeleton of $\mathcal{I}^*$.*

Returning to the (d, R)-subdivision task, we can now justify its name, simply by recalling that its output complex is equal to the algorithm complex:

Lemma 20 *The (d, R)-subdivision task over $\mathcal{I}^*$ for n processes is a triple $(\mathcal{I}^*, \mathcal{O}^*, \Delta^*)$, where $\mathcal{O}^*$ is the R-iterated, d-dimensional subdivision of the $n-1$ skeleton of $\mathcal{I}^*$, and Δ^* is equal to the corresponding subdivision carrier map.*

5.4 What Can Be Computed in the Presence of Solo Executions?

This section presents a characterization of the colorless tasks that can be solved in each one of the d-solo models.

Consider an r round colorless algorithm that solves the colorless task $(\mathcal{I}^*, \mathcal{O}^*, \Delta^*)$. At the end of the rth round, processes have to decide an output value, by executing $\mathsf{dec\_val}(\ell s_i)$ in line 3. The result of $\mathsf{dec\_val}(\ell s_i)$ is a vertex in $\mathcal{O}^*$, and different processes may decide different vertices as long as they belong to the same simplex of $\mathcal{O}^*$. This means that $\mathsf{dec\_val}$ is a simplicial map from $\mathcal{P}_r^*$ to $\mathcal{O}^*$. Moreover, $\mathsf{dec\_val}$ is carried by Δ^*, in the sense that for $\sigma \in \mathcal{I}^*$: $\mathsf{dec\_val}(\mathcal{P}_r^*(\sigma)) \subseteq \Delta^*(\sigma)$, which means that for any input simplex σ, any r round execution ends in a simplex τ of $\mathcal{P}_r^*$, and the decision that the processes make in τ, form an output simplex $\mathsf{dec\_val}(\tau)$ of $\mathcal{O}^*$. This output simplex $\mathsf{dec\_val}(\tau)$ must be in $\Delta^*(\sigma)$, to satisfy the task's specification.

Theorem 20 *The colorless task $\mathcal{T}^* = (\mathcal{I}^*, \mathcal{O}^*, \Delta^*)$ is solvable with n processes in the d-solo model by a colorless algorithm if and only if there is an $R \geq 0$ and a simplicial map $\phi : Div_d^R\ Skel^{n-1}\mathcal{I}^* \to \mathcal{O}^*$ carried by Δ^*.*

Proof (Sketch) If $\mathcal{T}^*$ is solvable in the d-solo model then there exists the simplicial map $\phi : \text{Div}_d^R\ \text{Skel}^{n-1}\mathcal{I}^* \to \mathcal{O}^*$ carried by Δ^*, by Lemma 18 and Lemma 19.

Conversely, notice that Lemma 20 says that the (d, R)-subdivision task over $\mathcal{I}^*$ for n processes has as output complex the R-iterated, d-dimensional subdivision of the $n-1$ skeleton of $\mathcal{I}^*$. By definition there is a colorless algorithm that solves this task. Thus, when each process p_i runs this algorithm, on an input simplex σ of $\mathcal{I}^*$, it gets as output a vertex v_i in $\text{Div}_d^R\text{Skel}^{n-1}\mathcal{I}^*$. Then p_i produces as output to $\mathcal{T}^*$ the value $\phi(v_i)$. Since the outputs v_i span a simplex of $\text{Div}_d^R\text{Skel}^{n-1}\mathcal{I}^*$, the outputs $\phi(v_i)$ span a simplex τ of $\mathcal{O}^*$, and τ is in $\Delta^*(\sigma)$. $\square_{Theorem\ 20}$

5.5 (d, ϵ)-Solo Approx. Agreement and Strict Hierarchy of Models

We now study the properties of the (d, R)-agreement task of Section 5.3.3 in terms of a precision parameter ϵ, showing that this task can be solved in the d-solo model while it cannot be solved in the $(d+1)$-solo model.

Let ϵ be a positive real. The (d, ϵ)-solo approximate agreement problem (in short (d, ϵ)-SAA) is a generalization of the *ϵ-approximate agreement* problem [27]. The $(1, \epsilon)$-solo approximate agreement instance implies 2ϵ-approximate agreement. Assuming the input of each process is a point of the d-dimensional Euclidean space $\mathbb{R}^d$, (d, ϵ)-solo approximate agreement is defined by the following properties.

- Validity. Any output lies within the convex hull of the inputs.
- Agreement. There is a set of processes S, $1 \leq |S| \leq d$, such that any process p_i that is not in S decides a value o_i (point) such that the Euclidean distance between o_i and CH is at most ϵ, where CH is the convex hull of the points decided by the processes in S.
- Termination. If a process p_i does not crash, it decides a value.

It follows from this definition that up to d processes are allowed to decide any set of points within the convex hull (as an example each of them may decide the point it proposes). These processes define the set S, and intuitively, the values they decide are collectively "represented" by their convex hull CH. Finally, the values decided by the other processes are constrained by the values decided by the processes in S.

The next theorem shows that, from a task solvability point of view, the d-solo model is stronger than the $(d+1)$-solo model.

Lemma 21 *If the volume of any d-face of the input complex is less than V, and $R > \frac{\log(V)+\log(d!)-d\log(\epsilon)}{\log(d+1)}$, then the (d, R)-agreement task solves the (d, ϵ)-solo approximate agreement problem.*

The proof shows that the smallest height of any d-simplex after R subdivisions is less than ϵ, which entails that, for any $(d+1)$ distinct values decided in the same execution, one is closer than ϵ from the $(d-1)$-face (convex hull) formed by the d other values.

Proof The validity property comes directly from the fact that, the values decided in the (d, R)-agreement task are barycenters of a subset of the input values. The termination follows from the fact that any correct process decides in R rounds.

Let us consider $\mathcal{P}_R^*$ the R-round complex of the colorless algorithm. If we show that the minimal height of any d-face of $\mathcal{P}_R^*$ is less than ϵ then the agreement property of the (d, ϵ)-solo approximate agreement problem follows.

Since, during each subdivision and in each d-face, a cone is built over the boundary with apex at the barycenter, the volume of the d-faces is multiplied by $\frac{1}{d+1}$ during each subdivision. It follows that, after R subdivisions with $V \cdot (\frac{1}{d+1})^R < \frac{\epsilon^d}{d!}$, the volume of any d-face of $\mathcal{P}_R^*$ is less than $\frac{\epsilon^d}{d!}$.

Let h_{min}^d be the smallest height of a d-face σ of $\mathcal{P}_R^*$, it is the distance between a vertex v_d of σ and the $(d-1)$-face of σ with the largest volume V_{max}^{d-1}. The volume of σ is $V(\sigma) = \frac{1}{d} h_{min}^d \cdot V_{max}^{d-1} < \frac{\epsilon^d}{d!}$. Consequently either $h_{min}^d < \epsilon$ or $V_{max}^{d-1} < \frac{\epsilon^{d-1}}{(d-1)!}$. In the first case the smallest height of σ is less than ϵ. In the second case, if $d-1 = 1$ then $V_{max}^1 < \epsilon$ is the largest distance between to vertices of σ and then the smallest height of σ is less than ϵ. If $d-1 > 1$ then consider the smallest height h_{min}^{d-1} of $\sigma \setminus \{v_d\}$ whose length is the distance between a vertex v_{d-1} of $\sigma \setminus \{v_d\}$ and the face of $\sigma \setminus \{v_d\}$ with the largest volume V_{max}^{d-2}. Since h_{min}^{d-1} is smaller than the distance between v_{d-1} and its complementary face in σ, $h_{min}^{d-1} \geq h_{min}^d$. The volume of $\sigma \setminus \{v_d\}$ is $V_{max}^{d-1} = \frac{1}{d-1} h_{min}^{d-1} \cdot V_{max}^{d-2} < \frac{\epsilon^{d-1}}{(d-1)!}$, consequently either $h_{min}^d \leq h_{min}^{d-1} < \epsilon$ or we can iterate with $V_{max}^{d-2} < \epsilon^{d-2}(d-2)!$. $\square_{Lemma\ 21}$

Lemma 22 *For $d > 1$, if the domain of the possible inputs contains a regular simplex of dimension $(d-1)$ whose edge length is strictly greater than $2\epsilon(d-1)\sqrt{\frac{2(d-1)}{d}}$, then, for $n \geq d$, the $(d-1, \epsilon)$-solo approximate agreement problem is impossible to solve in $\mathcal{ACS}^d$.*

Proof The proof is by contradiction. Assuming that there is an algorithm A that solves $(d-1, \epsilon)$-SAA in $\mathcal{ACS}^d$, let us consider its executions in which the processes p_{d+1}, ..., p_n crash before executing any step, and each process p_i, $1 \leq i \leq d$ retrieves only its own value from the CO^d bogey's in every round. As no more than d processes invoke A, there is a subset of executions in which the behavior of each process p_i, $1 \leq i \leq d$, is indistinguishable from a solo execution.

Hence, to show that $(d-1, \epsilon)$-SAA cannot be solved in $\mathcal{ACS}^d$, let the values v_1, ..., v_d proposed to $(d-1, \epsilon)$-SAA by the processes p_1, ..., p_d be d distinct vertices of a regular simplex of dimension $(d-1)$ whose the edge length α is such that $\alpha > 2\epsilon(d-1)\sqrt{\frac{2(d-1)}{d}}$ (this is possible since, by hypothesis, the domain of possible inputs contains such a simplex). In the following $dist(a, X)$ denotes the Euclidean distance between a and X, where a is a vertex, and X is a vertex, a polytope, or a hyperplane. It follows from the termination, agreement and validity properties of A that there exists a plane $\mathcal{P}$ of dimension $(d-2)$ such that $\forall\ i \in \{1, \ldots, d\} : dist(v_i, \mathcal{P}) \leq \epsilon$. This is a consequence of the agreement property. (Up to $d-1$ processes can decide any value. The convex hull of this set of values is a polytope P' of dimension $(d-2)$ or less and all the other processes decide either one of the previous values or a value distant of at most ϵ from the barycenter of these values. In both cases their values are distant of at most ϵ from P'. Since the dimension of P' is at most $d-2$, any $(d-2)$-plane $\mathcal{P}$ containing P' verifies the property.)

Remark that, since they are the vertices of a regular $(d-1)$-dimensional simplex, the points v_1, ..., v_d do not belong to the same plane of dimension $d-2$ (hence, the vectors $v_1v_2, \ldots, v_1v_d$ are linearly independent).

Let N_i be a normalized vector, orthogonal to the hyperplane $\mathcal{P}_i$ that contains the points $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_d$. Then, $\forall i \in [2, d]$, $dist(v_i, \mathcal{P}_i) = |\langle N_i | v_1 v_i \rangle|$ (where $|\langle N_i | v_1 v_i \rangle|$ denotes the scalar product of the vector N_i by the vector $v_1 v_i$).

Let B_i be the d balls of center v_i and radius ϵ. Since $\forall i$, $dist(v_i, P) \leq \epsilon$, there exist d vectors $\epsilon_1, \ldots, \epsilon_d$ such that the points $v'_1 = v_1 + \epsilon_1, \ldots, v'_d = v_d + \epsilon_d \in B_1, \ldots, B_d$ and $\forall\ i$, $v'_i \in$

P. The points $v'_1, \ldots, v'_d$ are coplanar and the vectors $v'_1v'_2, \ldots, v'_1v'_d$ are not independent, and consequently there exist $d-1$ real numbers $\lambda_2, \ldots, \lambda_d \neq 0, \ldots, 0$ such that $\sum_{i=2}^{d} \lambda_i v'_1 v'_i = 0$.

Let λ_j be such that $|\lambda_j| = \max_i |\lambda_i|$. We have then

$$0 = \langle N_j | \sum_{i=2}^{d} \lambda_i v'_1 v'_i \rangle, \tag{5.1}$$

$$= \langle N_j | \sum_{i=2}^{d} \lambda_i (v_1 v_i + \epsilon_i - \epsilon_1) \rangle, \tag{5.2}$$

$$= \lambda_j \langle N_j | v_1 v_j \rangle + \sum_{i=2}^{d} \lambda_i \langle N_j | (\epsilon_i - \epsilon_1) \rangle. \tag{5.3}$$

Thus,

$$\lambda_j \langle N_j | v_1 v_j \rangle = - \sum_{i=2}^{d} \lambda_i \langle N_j | (\epsilon_i - \epsilon_1) \rangle, \tag{5.4}$$

$$|\lambda_j \langle N_j | v_1 v_j \rangle| = |\sum_{i=2}^{d} \lambda_i \langle N_j | (\epsilon_i - \epsilon_1) \rangle|, \tag{5.5}$$

$$|\lambda_j| dist(v_j, \mathcal{P}_j) = |\sum_{i=2}^{d} \lambda_i \langle N_j | (\epsilon_i - \epsilon_1) \rangle|. \tag{5.6}$$

But

$$|\sum_{i=2}^{d} \lambda_i \langle N_j | (\epsilon_i - \epsilon_1) \rangle| \leq \sum_{i=2}^{d} |\lambda_i| (|\epsilon_i| + |\epsilon_1|), \tag{5.7}$$

$$\leq |\lambda_j| \times 2\epsilon \cdot (d-1), \tag{5.8}$$

which would imply that $dist(v_j, \mathcal{P}_j) \leq 2\epsilon \cdot (d-1)$, while, according to the definition of the points v_i, $dist(v_j, \mathcal{P}_j) = \alpha\sqrt{\frac{d}{2(d-1)}} > 2\epsilon \cdot (d-1)$. ($\alpha\sqrt{\frac{d}{2(d-1)}}$ is the height of a regular $d-1$-simplex of edge size α.) This contradicts the existence of $\mathcal{P}$ and thus the existence of the algorithm A. $\square_{Lemma\ 22}$

Theorem 21 *If the domain of the possible input values (a) is bounded and (b) contains a regular simplex of dimension d whose edge length is strictly greater than $2\epsilon d\sqrt{\frac{2d}{d+1}}$, then the (d, ϵ)-solo approximate agreement problem is solvable in $\mathcal{ACS}^d$ but not in $\mathcal{ACS}^{d+1}$.* (Follows directly from the two previous lemmas.)

5.6 On Approximate Agreement

5.6.1 On the agreement property of (d, ϵ)-solo approximate agreement

As in all approximate agreement problems (see below) the idea is to force processes to decide values that are not too far the ones from the others. But, as each process that executes solo sees only the value it proposes, it can decide only its value. Differently, the other processes see the values decided by the processes which executed solo. This motivates the definition given in Section 5.5.

More precisely, the fact that up to d processes are allowed to decide any values in the convex hull of the proposed values is directly related to the possibility of up to d processes executing solo in an execution. This set of at most d processes defines the set S used in the definition of (d, ϵ)-SAA. As indicated, when this occurs, the values decided by these processes are collectively represented by their convex hull CH and any other process has to decide a value "not too far" from CH.

If no process executes solo, the agreement property states that the set S has nevertheless to contain at least one process: $1 \leq |S| \leq d$ (hence S has not to be confused with the operational set π_1 used in the definition of the communication objects involved in the d-solo model, Section 5.2.1). As the values decided by the processes in S are then within the convex hull of the proposed values, it is as these processes have executed solo.

Remark One could think to have an agreement property composed of two parts, defined as follows:

- Solo execution agreement. If at least one process executes solo, the agreement property is the one described in Section 5.5.
- No-solo execution agreement. If no process executes solo, the Euclidean distance between any two decided values is at most ϵ.

Unfortunately, as in the non-blocking atomic commit problem (NBAC), this definition involves the behavior of the run, which becomes an input of the problem. It follows that, as NBAC, the problem captured by this extended definition is not a task (a task is defined by an application from input vectors to output vectors and this application has to be independent of the execution pattern). *End of remark.*

5.6.2 Relating d-Set Agreement and (d, ϵ)-Solo Approx. Agreement

The d-set agreement (in short d-SA) problem [22] is defined as follows. Assuming that every process proposes a value, each process that does not crash has to decide a value (termination), such that a decided value is a proposed value (validity), and at most d different values are decided (agreement). Similarly to ϵ-approximate agreement which is a weakened version of consensus, (d, ϵ)-SAA is a weakened version of d-SA.

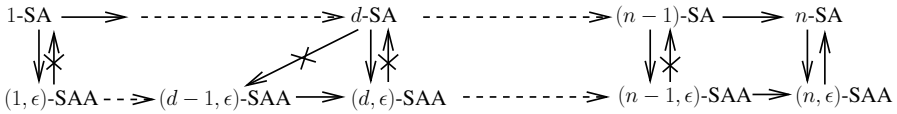


Figure 5.4: Relating d-SA with both (d, ϵ)-SAA and $(d-1, \epsilon)$-SAA

Considering the wait-free asynchronous message-passing model enriched with an algorithm solving d-SA (denoted $\mathcal{AMP}[d{-}\mathrm{SA}]$ in the following), this section shows that (d, ϵ)-SAA can be solved in this model while $(d-1, \epsilon)$-SAA cannot. It also shows that d-SA cannot be solved in $\mathcal{ACS}^d$ (which is stronger than $\mathcal{AMP}[d{-}\mathrm{SA}]$). The resulting computability map is represented in Figure 5.4. An arrow means that a reduction exists, while a crossed out arrow means that no reduction exists. (Let us observe that the arrows for $d = n$ are trivial as, in this case, each process is allowed to decide the value it proposes without communicating with other processes.)

Theorem 22 *It is possible to solve (d, ϵ)-SAA in $\mathcal{AMP}[d{-}\mathrm{SA}]$.*

Proof Let the input of each process p_i be the d-solo coordinates of a point in $\mathbb{R}^d$. The processes execute first a d-set agreement algorithm, at the end of which they agree on at most d points of

$\mathbb{R}^d$. As these points have been proposed by processes, they belong to the convex hull of proposed values, and consequently satisfy the validity property of (d, ϵ)-SAA. Moreover, as no more than d different points are output by d-set agreement algorithm, they trivially satisfy the agreement property of (d, ϵ)-SAA, which concludes the proof. $\square_{Theorem\ 22}$

Theorem 23 *It is impossible to solve $(d-1, \epsilon)$-SAA in $\mathcal{ACS}^d[d{-}\mathrm{SA}]$.*

Let us notice that, as $\mathcal{ACS}^d[d{-}\mathrm{SA}]$ is stronger than $\mathcal{AMP}[d{-}\mathrm{SA}]$, it follows from this theorem that $(d-1, \epsilon)$-solo approximate agreement cannot be solved in $\mathcal{AMP}[d{-}\mathrm{SA}]$.

Proof The proof is by contradiction. Assuming that there is an algorithm A that solves $(d-1, \epsilon)$-SAA in $\mathcal{ACS}^d[d{-}\mathrm{SA}]$, let us consider its executions in which the processes p_{d+1}, ..., p_n crash before executing any step, and the messages among the processes p_1, ..., p_d are delayed until each of them has decided. each process p_i, $1 \leq i \leq d$ retrieves only its own value from the CO^d objects in every round. As no more than d processes invoke A, there is a subset of executions in which the behavior of each process p_i, $1 \leq i \leq d$, is indistinguishable from a solo execution. Let us observe that, in these executions of A, each process can obtain from the underlying d-SA algorithm the value it has proposed to it (i.e., in these executions, the d-SA algorithm provides none of the processes p_1, ..., p_d with new information).

The rest of the proof is then the same as the proof of Lemma 22, starting now after its first paragraph by the sentence "Hence, to show that $(d-1, \epsilon)$-SAA cannot be solved in $\mathcal{AMP}[d{-}\mathrm{SA}]$, let the values v_1, ..., v_d", etc. $\square_{Theorem\ 23}$

Theorem 24 *For $d < n$, it is impossible to solve d-set agreement in $\mathcal{ACS}^d$.*

Proof Let us first observe that $\mathcal{ACS}^d$ can be simulated in $\mathcal{ARW}$. Hence, if d-SA can be solved in $\mathcal{ACS}^d$, it can also be solved in $\mathcal{ARW}$. But this contradicts the theorem stating that it is impossible to solve d-SA in $\mathcal{ARW}$ [13, 45, 89], which completes the proof. $\square_{Theorem\ 24}$

5.7 Conclusion and Remarks on the d-Solo Models and their Properties

In this chapter we introduced a new family of asynchronous wait-free models bridging the gap between $\mathcal{AMP}$, in which all the processes may have to decided only from their own input as if they were executing alone, and $\mathcal{ARW}$, in which only one process can be in this situation. We showed that the colorless tasks that can be computed by the colorless algorithm in these intermediary models can be characterized through topology using a similar approach as in [45]. A new family of problems, denoted (d, ϵ)-solo approximate agreements has also been introduced and used to prove that there are tasks that can be computed in the d-solo model but not in the $(d+1)$-solo model. In consequence, the hierarchy formed by these new models is strict in terms of computability. Finally these new problems have been compared to the k-set agreement problem.

To extend this work, several possibilities may be considered. It could be interesting to redefine the communication object used in the d-solo models in order to allow more behaviors, for example by letting the processes of the second set of the set-linearization of the accesses to CO^d to miss the writes of the others and part of the writes that happened first. It would allow rounds such that, for example, p_1 and p_2 are set-linearized first and each of them only sees its own value, while p_3 and p_4 are set-linearized in the second position, p_3 only seeing its own value and the value written

by p_1 while p_4 sees its own value and the one written by p_2. Like that we could more completely represent partitioning systems by an iterated model while still being able to restrict the number of partitions. Other additions to this work could consist in extending the topological study of these models to colorful algorithms, or trying to study decidability through loop agreement as it has been done in the case of $\mathcal{IIS}$.

Chapter 6

Conclusion and Perspectives

6.1 Topics Covered in this Thesis

In this document, several abstractions, problems, algorithms and reductions have been introduced in order to better understand, in terms of computability, the links between failure detectors, partitioning systems and iterated models.

k-Set agreement and s-simultaneous consensus The k-set agreement problem and its parallel variant, the s-simultaneous consensus, that may be seen as the counterparts of the consensus in partitioning systems had a central role in these works. Constructions that can be based on them have been presented in Chapter 4. Chapter 3 studied their difference when the communication is based on message-passing and presented failure detectors suited to them.

Failure detectors The failure detector approach that allows to build modular algorithms for failure-prone asynchronous systems and to compare problems in a given system has been extended to two iterated models in Chapter 2. Failure detectors have also been instrumental in the study of the hierarchy of problems that emerged from the difference between the k-set agreement and the s-simultaneous consensus when considered in the asynchronous message-passing model in Chapter 3. However, the question of the weakest failure detector to solve k-set agreement in this model is still open.

Partitioning systems Systems prone to partitioning have been studied under several angles in this document. Chapter 4 proposed a modular construction to implement shared objects based on simultaneous consensus objects. Since these objects may be possible to implement even in presence of partitions if a suitable failure detector is available, it is possible to imagine adapting the proposed construction to implement distributed objects in these partition-prone systems, for example on a platform distributed across several datacenters or cloud providers, if the application using them can tolerate the implied sacrifices in terms of liveness. The theoretical distinction between k-set agreement and s-simultaneous consensus presented in Chapter 3 may also have implications on the design of such platform, since it underlines that being able to solve only the k-set agreement or both of them does not provide a system with the same computational abilities and there are refinement levels. In addition of that, Chapter 5 introduces a family of iterated models in which a controlled number of processes can run in isolation, which can be viewed as the first step toward partitioning. These models admit a clean topological representation of the set of possible executions, opening the way for deeper theoretical understanding of their computability properties.

Iterated models The study of the constrained structure of the communication in iterated models made us define the notion of strongly correct processes in Chapter 2. These processes, that have the possibility to communicate, infinitely often, directly or not, with all the other, play an important role in the simulations between the studied iterated models and the non-iterated asynchronous models enriched with failure detectors. Chapter 5 also proposed a new family of iterated models in which several processes can run independently. According to the maximum number of processes allowed to run without receiving any communication from the others, the tasks that can be computed in these models vary from what can be computed in a wait-free manner with an asynchronous shared memory to the few tasks wait-free computable in an asynchronous message-passing model. The structure of the possible executions in these models has a simple topological representation.

6.2 Other Publications During this PhD

Besides the studies presented in this document, namely [81, 85, 84, 87, 43], the work done during this PhD has covered several other topics.

The quest for the weakest failure detector for the k-set agreement problem in asynchronous message-passing systems lead to a study of the relations between the existing failure detectors for this problem [66] and to the proposal of a new failure detector suited to that problem and weaker than the previously proposed ones [67]. The failure detector used in Chapter 3 is based on this failure detector.

Two Ω-based consensus algorithms have been proposed, one using closing sets [82] and another relying on store-collect objects [83]. An algorithm for the broadcast problem in recurrent dynamic systems in which channels are not reliable has been published in [86]. In the case of asynchronous message-passing systems prone to Byzantine failures, an algorithm allowing the construction of reliable atomic shared registers has been published in [47]. The way groups can be used to reduce the number of names for the renaming problem in asynchronous shared memory systems has also been studied in [18]. Finally, I collaborated on the journal version of a paper that proposes an approach to leverage the trust expressed by explicit social networks in order to build a random peer sampling protocol for peer-to-peer systems. This work appears in [32].

6.3 Perspectives

In the continuation of this thesis, I plan to try to further my exploration of the challenges that arise in systems prone to combinations of asynchrony and failures. I am interested in the path opened by the notion of message-adversary [2] and its ability to capture in message losses the two other sources of uncertainty that are the asynchrony and the crashes. I also would like to study models that allow to express correlations between failures such as the core and survivor sets of [49]. I am also working on trying to improve the approach presented in [47] that aims at providing reliable abstractions to help designing algorithms on top of asynchronous systems prone to Byzantine failures. Finally, I still have hope that the weakest failure detector to solve the k-set agreement in an asynchronous message-passing systems will be found.

Bibliography

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, Nir Shavit: Atomic Snapshots of Shared Memory. J. ACM 40(4): 873-890 (1993)

[2] Yehuda Afek, Eli Gafni: Asynchrony from Synchrony. ICDCN 2013: 225-239

[3] Yehuda Afek, Eli Gafni, and Nati Linial: A king in two tournaments. Unpublished report 3 (2012).

[4] Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, Corentin Travers: The k-simultaneous consensus problem. Distributed Computing 22(3): 185-195 (2010)

[5] Emmanuelle Anceaume, Antonio Fernández, Achour Mostéfaoui, Gil Neiger, Michel Raynal: A necessary and sufficient condition for transforming limited accuracy failure detectors. J. Comput. Syst. Sci. 68(1): 123-133 (2004)

[6] James H. Anderson, Mark Moir: Universal Constructions for Large Objects. IEEE Trans. Parallel Distrib. Syst. 10(12): 1317-1332 (1999)

[7] Antonio Fernández Anta, Michel Raynal: From an Asynchronous Intermittent Rotating Star to an Eventual Leader. IEEE Trans. Parallel Distrib. Syst. 21(9): 1290-1303 (2010)

[8] Hagit Attiya, Amotz Bar-Noy, Danny Dolev: Sharing Memory Robustly in Message-Passing Systems. J. ACM 42(1): 124-142 (1995)

[9] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, Rüdiger Reischuk: Renaming in an Asynchronous Environment. J. ACM 37(3): 524-548 (1990)

[10] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov: The complexity of obstruction-free implementations. J. ACM 56(4) (2009)

[11] Hagit Attiya, Jennifer Welch: Distributed computing: fundamentals, simulations, and advanced topics. John Wiley & Sons 2004.

[12] François Bonnet, Michel Raynal: On the road to the weakest failure detector for k-set agreement in message-passing systems. Theor. Comput. Sci. 412(33): 4273-4284 (2011)

[13] Elizabeth Borowsky, Eli Gafni: Generalized FLP impossibility result for t-resilient asynchronous computations. STOC 1993: 91-100

[14] Elizabeth Borowsky, Eli Gafni: A Simple Algorithmically Reasoned Characterization of Wait-Free Computations (Extended Abstract). PODC 1997: 189-198

[15] Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, Sergio Rajsbaum: The BG distributed simulation algorithm. Distributed Computing 14(3): 127-146 (2001)

[16] Zohir Bouzid, Corentin Travers: (anti-Omegax x Sigmaz)-Based k-Set Agreement Algorithms. OPODIS 2010: 189-204

[17] Zohir Bouzid, Corentin Travers: Parallel Consensus is Harder than Set Agreement in Message Passing. ICDCS 2013: 611-620

[18] Armando Castañeda, Michel Raynal, Julien Stainer: When and How Process Groups Can Be Used to Reduce the Renaming Space. OPODIS 2012: 91-105

[19] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg: The Weakest Failure Detector for Solving Consensus. J. ACM 43(4): 685-722 (1996)

[20] Tushar Deepak Chandra, Sam Toueg: Unreliable Failure Detectors for Reliable Distributed Systems. J. ACM 43(2): 225-267 (1996)

[21] Bernadette Charron-Bost, André Schiper: The Heard-Of model: computing in distributed systems with benign faults. Distributed Computing 22(1): 49-71 (2009)

[22] Soma Chaudhuri: More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. Inf. Comput. 105(1): 132-158 (1993)

[23] Alejandro Cornejo, Sergio Rajsbaum, Michel Raynal, Corentin Travers: Failure detectors are schedulers. PODC 2007: 308-309

[24] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui: Tight failure detection bounds on atomic object implementations. J. ACM 57(4) (2010)

[25] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, Sam Toueg: The weakest failure detectors to solve certain fundamental problems in distributed computing. PODC 2004: 338-346

[26] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Andreas Tielmann: The Weakest Failure Detector for Message Passing Set-Agreement. DISC 2008: 109-120

[27] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, William E. Weihl: Reaching approximate agreement in the presence of faults. J. ACM 33(3): 499-516 (1986)

[28] Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, Corentin Travers: Universal constructions that ensure disjoint-access parallelism and wait-freedom. PODC 2012: 115-124

[29] Tzilla Elrad, Nissim Francez: Decomposition of Distributed Programs into Communication-Closed Layers. Sci. Comput. Program. 2(3): 155-173 (1982)

[30] Panagiota Fatourou, Nikolaos D. Kallimanis: Highly-Efficient Wait-Free Synchronization. Theory Comput. Syst. 55(3): 475-520 (2014)

[31] Michael J. Fischer, Nancy A. Lynch, Mike Paterson: Impossibility of Distributed Consensus with One Faulty Process. J. ACM 32(2): 374-382 (1985)

[32] Davide Frey, Arnaud Jégou, Anne-Marie Kermarrec, Michel Raynal, Julien Stainer: Trust-aware peer sampling: Performance and privacy tradeoffs. Theor. Comput. Sci. 512: 67-83 (2013)

[33] Eli Gafni: Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony (Extended Abstract). PODC 1998: 143-152

[34] Eli Gafni, Rachid Guerraoui: Generalized Universality. CONCUR 2011: 17-27

[35] Eli Gafni, Petr Kuznetsov: On set consensus numbers. Distributed Computing 24(3-4): 149-163 (2011)

[36] Eli Gafni, Sergio Rajsbaum: Distributed Programming with Tasks. OPODIS 2010: 205-218

[37] Rachid Guerraoui, Michel Raynal: The Alpha of Indulgent Consensus. Comput. J. 50(1): 53-67 (2007)

[38] Maurice Herlihy: Wait-Free Synchronization. ACM Trans. Program. Lang. Syst. 13(1): 124-149 (1991)

[39] Maurice Herlihy, Dmitry N. Kozlov, Sergio Rajsbaum: Distributed Computing Through Combinatorial Topology. Morgan Kaufmann 2013, ISBN 978-0-12-404578-1

[40] Maurice Herlihy, Victor Luchangco, Mark Moir: Obstruction-Free Synchronization: Double-Ended Queues as an Example. ICDCS 2003: 522-529

[41] Maurice Herlihy, Sergio Rajsbaum: A classification of wait-free loop agreement tasks. Theor. Comput. Sci. 291(1): 55-77 (2003)

[42] Maurice Herlihy, Sergio Rajsbaum: The topology of shared-memory adversaries. PODC 2010: 105-113

[43] Maurice Herlihy, Sergio Rajsbaum, Michel Raynal, Julien Stainer: Computing in the Presence of Concurrent Solo Executions. LATIN 2014: 214-225

[44] Maurice Herlihy, Sergio Rajsbaum, Michel Raynal, Julien Stainer: Computing in the Presence of Concurrent Solo Executions. Technical Report: https://hal.inria.fr/hal-00825619

[45] Maurice Herlihy, Nir Shavit: The topological structure of asynchronous computability. J. ACM 46(6): 858-923 (1999)

[46] Maurice Herlihy, Jeannette M. Wing: Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst. 12(3): 463-492 (1990)

[47] Damien Imbs, Sergio Rajsbaum, Michel Raynal, Julien Stainer: Reliable Shared Memory Abstraction on Top of Asynchronous Byzantine Message-Passing Systems. SIROCCO 2014: 37-53

[48] Prasad Jayanti, Sam Toueg: Every problem has a weakest failure detector. PODC 2008: 75-84

[49] Flavio Paiva Junqueira, Keith Marzullo: Designing Algorithms for Dependent Process Failures. Future Directions in Distributed Computing 2003: 24-28

[50] Fabian Kuhn, Nancy A. Lynch, Rotem Oshman: Distributed computation in dynamic networks. STOC 2010: 513-522

[51] Leslie Lamport: On Interprocess Communication. Part I: Basic Formalism. Distributed Computing 1(2): 77-85 (1986)

[52] Leslie Lamport: On Interprocess Communication. Part II: Algorithms. Distributed Computing 1(2): 86-101 (1986)

[53] Leslie Lamport: The Part-Time Parliament. ACM Trans. Comput. Syst. 16(2): 133-169 (1998)

[54] Leslie Lamport, Robert E. Shostak, Marshall C. Pease: The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. 4(3): 382-401 (1982)

[55] Edmund Georg Hermann Landau: On dominance relations and the structure of animal societies: III The condition for a score structure. Bulletin of Mathematical Biology 15(2): 143-148 (1953)

[56] Wai-Kau Lo, Vassos Hadzilacos: Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems (Extended Abstract). WDAG 1994: 280-295

[57] Michael C. Loui, Hosame H. Abu-Amara: Memory requirements for agreement among unreliable asynchronous processes. Advances in Computing research 4: 163-183 (1987)

[58] Victor Luchangco, Mark Moir, Nir Shavit: On the Uncontended Complexity of Consensus. DISC 2003: 45-59

[59] Nancy A. Lynch: Distributed Algorithms. Morgan Kaufmann 1996, ISBN 1-55860-348-4

[60] Hammurabi Mendes, Maurice Herlihy: Multidimensional approximate agreement in Byzantine asynchronous systems. STOC 2013: 391-400

[61] Michael Merritt, Gadi Taubenfeld: Resilient Consensus for Infinitely Many Processes. DISC 2003: 1-15

[62] Achour Mostéfaoui, Eric Mourgaya, Michel Raynal: Asynchronous Implementation of Failure Detectors. DSN 2003: 351-360

[63] Achour Mostéfaoui, Sergio Rajsbaum, Michel Raynal: Conditions on input vectors for consensus solvability in asynchronous distributed systems. J. ACM 50(6): 922-954 (2003)

[64] Achour Mostéfaoui, Michel Raynal: Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: A General Quorum-Based Approach. DISC 1999: 49-63

[65] Achour Mostéfaoui, Michel Raynal: k-set agreement with limited accuracy failure detectors. PODC 2000: 143-152

[66] Achour Mostéfaoui, Michel Raynal, Julien Stainer: Relations Linking Failure Detectors Associated with k-Set Agreement in Message-Passing Systems. SSS 2011: 341-355

[67] Achour Mostéfaoui, Michel Raynal, Julien Stainer: Chasing the Weakest Failure Detector for k-Set Agreement in Message-Passing Systems. NCA 2012: 44-51

[68] James R. Munkres: Elements of algebraic topology. Addison-Wesley 1984, ISBN 978-0-201-04586-4, pp. I-IX, 1-454

[69] Gil Neiger: Set-Linearizability. PODC 1994: 396

[70] Gil Neiger: Failure Detectors and the Wait-Free Hierarchy. PODC 1995: 100-109

[71] Marshall C. Pease, Robert E. Shostak, Leslie Lamport: Reaching Agreement in the Presence of Faults. J. ACM 27(2): 228-234 (1980)

[72] Scott M. Pike, Srikanth Sastry, Jennifer L. Welch: Failure detectors encapsulate fairness. Distributed Computing 25(4): 313-333 (2012)

[73] Sergio Rajsbaum, Michel Raynal, Corentin Travers: An impossibility about failure detectors in the iterated immediate snapshot model. Inf. Process. Lett. 108(3): 160-164 (2008)

[74] Sergio Rajsbaum, Michel Raynal, Corentin Travers: The Iterated Restricted Immediate Snapshot Model. COCOON 2008: 487-497

[75] Michel Raynal: K-anti-Omega. Rump Session at PODC07

[76] Michel Raynal: Failure detectors for asynchronous distributed systems: an introduction. Wiley Encyclopedia of Computer Science and Engineering 2: 1181-1191 (2009)

[77] Michel Raynal: Fault-tolerant Agreement in Synchronous Message-passing Systems. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers 2010

[78] Michel Raynal: Failure Detectors to Solve Asynchronous k-Set Agreement: a Glimpse of Recent Results. Bulletin of the EATCS 103: 74-95 (2011)

[79] Michel Raynal: Distributed Algorithms for Message-Passing Systems. Springer 2013, ISBN 978-3-642-38122-5, pp. I-XXX, 1-500

[80] Michel Raynal: Concurrent Programming - Algorithms, Principles, and Foundations. Springer 2013, ISBN 978-3-642-32026-2, pp. I-XXXII, 1-515

[81] Michel Raynal, Julien Stainer: Increasing the Power of the Iterated Immediate Snapshot Model with Failure Detectors. SIROCCO 2012: 231-242

[82] Michel Raynal, Julien Stainer: A Simple Asynchronous Shared Memory Consensus Algorithm Based on Omega and Closing Sets. CISIS 2012: 357-364

[83] Michel Raynal, Julien Stainer: From a Store-Collect Object and Ω to Efficient Asynchronous Consensus. Euro-Par 2012: 427-438

[84] Michel Raynal, Julien Stainer: Simultaneous Consensus vs Set Agreement: A Message-Passing-Sensitive Hierarchy of Agreement Problems. SIROCCO 2013: 298-309

[85] Michel Raynal, Julien Stainer: Synchrony weakened by message adversaries vs asynchrony restricted by failure detectors. PODC 2013: 166-175

[86] Michel Raynal, Julien Stainer, Jiannong Cao, Weigang Wu: A Simple Broadcast Algorithm for Recurrent Dynamic Systems. AINA 2014: 933-939

[87] Michel Raynal, Julien Stainer, Gadi Taubenfeld: Distributed Universality. OPODIS 2014: 469-484

[88] Michel Raynal, Corentin Travers: In Search of the Holy Grail: Looking for the Weakest Failure Detector for Wait-Free Set Agreement. OPODIS 2006: 3-19

[89] Michael E. Saks, Fotios Zaharoglou: Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. SIAM J. Comput. 29(5): 1449-1483 (2000)

[90] Nicola Santoro, Peter Widmayer: Agreement in synchronous networks with ubiquitous faults. Theor. Comput. Sci. 384(2-3): 232-249 (2007)

[91] Nicola Santoro, Peter Widmayer: Time is Not a Healer. STACS 1989: 304-313

[92] Ulrich Schmid, Bettina Weiss, Idit Keidar: Impossibility Results and Lower Bounds for Consensus under Link Failures. SIAM J. Comput. 38(5): 1912-1951 (2009)

[93] Gadi Taubenfeld: Contention-Sensitive Data Structures and Algorithms. DISC 2009: 157-171

[94] Nitin H. Vaidya, Vijay K. Garg: Byzantine vector consensus in complete graphs. PODC 2013: 65-73

[95] Eric Christopher Zeeman: Relative simplicial approximation. Mathematical Proceedings of the Cambridge Philosophical Society. Vol. 60. No. 01. Cambridge University Press, 1964.

[96] Piotr Zielinski: Anti-Omega: the weakest failure detector for set agreement. Distributed Computing 22(5-6): 335-348 (2010)

List of Figures