Failure Detectors in Dynamic Distributed Systems

Élise Jeanneau

SORBONNE UNIVERSITÉ
Inria inventors for the digital world

École doctorale EDITE de Paris

Failure Detectors in Dynamic Distributed Systems

THÈSE

présentée et soutenue publiquement le 7 décembre 2018

pour l'obtention du

Doctorat de Sorbonne Université

(mention informatique)

par

Denis Jeanneau

Composition du jury

Rapporteurs :	Carole Delporte Roy Friedman
Examinateurs :	Sébastien Tixeuil Arnaud Casteigts
Encadrants :	Luciana Arantes Pierre Sens

Laboratoire d'Informatique de Paris 6 (LIP6)

Acknowledgements

I am first grateful to Lucana Arantes and Pierre Sens for their guidance, feedback and support during the three years of this thesis.

I would like to thank Carole Delporte, Roy Friedman, Arnaud Casteigts and Sébastien Tixeuil for accepting to be part of the jury for this thesis. Additionally, I want to thank Carole Delporte and Roy Friedman for accepting to review this manuscript.

Of course, I want to thank my co-authors, Thibault Rieutord, Etienne Mauffret, Luiz A. Rodrigues and Elias P. Duarte Jr., for their collaboration without which none of the publications in this thesis could have happened.

I would also like to thank all the colleagues at LIP6, which is a great work environment, and in particular the members of teams DELYS and Whisper for their friendship.

Finally, I want to thank my family for their moral support through this thesis and everything else.

Résumé

Dans le problème du consensus, tous les processus corrects du système doivent se mettre d'accord sur une unique valeur. Le problème du k-accord est une généralisation du consensus dans laquelle les processus peuvent se mettre d'accord sur au plus k valeurs. Le problème de l'exclusion mutuelle nécessite de permettre aux processus l'accès à une section critique, tout en garantissant qu'au plus un processus peut se trouver en section critique à tout moment. Ces problèmes fondamentaux de l'algorithmique distribuée ont été largement étudiés, mais la plupart des solutions existantes s'appuient sur l'hypothèse implicite que le système n'est pas dynamique.

Traditionellement, l'algorithmique distribuée consiste à étudier des systèmes distribués dans lesquels la liste des participants est statique et le graphe de communication est connecté, voire complet. Mais afin de modéliser les réseaux modernes, tels que les réseaux pair-à-pair ou sans fil, il est nécessaire de considérer une nouvelle sorte de systèmes distribués. Les systèmes dynamiques sont des systèmes distribués dans lesquels (1) les processus peuvent rejoindre ou quitter le système en cours d'exécution, et (2) le graphe de communication évolue au fil du temps.

L'abstraction des détecteurs de fautes a été introduite afin de contourner l'impossibilité de résoudre le consensus dans les systèmes asynchrones sujets aux pannes franches. Un détecteur de fautes est un oracle local fournissant aux processus des informations non fiables sur les pannes de processus. Mais un détecteur de fautes qui est suffisant pour résoudre un problème donné dans un système statique n'est pas nécessairement suffisant pour résoudre le même problème dans un système dynamique. De plus, certains détecteurs de fautes ne peuvent pas être implémentés dans un système dynamique. Par conséquent, il est nécessaire de redéfinir les détecteurs de fautes existants et de concevoir de nouveaux algorithmes.

Dans cette thèse, nous fournissons une nouvelle définition d'un détecteur de fautes pour le k-accord, et nous prouvons qu'il est suffisant pour résoudre le k-accord dans un système dynamique. Nous définissons également un modèle de système dynamique, ainsi qu'un algorithme capable d'implémenter ce nouveau détecteur de fautes dans notre modèle.

De plus, nous adaptons un détecteur existant pour l'exclusion mutuelle et nous prouvons que même dans les systèmes dynamiques, il s'agit toujours du détecteur de fautes le plus faible pour résoudre l'exclusion mutuelle. Cela signifie que ce détecteur est plus faible que tous les autres détecteurs capables de résoudre l'exclusion mutuelle.

Mots-clés: systèmes distribués, détecteurs de fautes, systèmes dynamiques, k-accord, exclusion mutuelle

Abstract

In the consensus problem, all correct processes in the system must agree on a same value. The k-set agreement problem is a generalization of consensus where processes can agree on up to k different values. The mutual exclusion problem requires processes to be able to access a critical section, such that no two processes can be in the critical section at the same time. These fundamental problems of distributed computing have been widely studied, but most existing solutions assume that the system is not dynamic.

Traditionally, distributed computing considers distributed systems where the system membership is static and the communication graph is connected or fully connected. But in order to model modern networks such as wireless or peer-to-peer networks, it is necessary to consider another kind of distributed systems. Dynamic systems are distributed systems in which (1) processes can join or leave the system during the run, and (2) the communication graph evolves over time.

The failure detector abstraction was introduced as a way to circumvent the impossibility of solving consensus in asynchronous systems prone to crash failures. A failure detector is a local oracle that provides processes in the system with unreliable information on process failures. But a failure detector that is sufficient to solve a given problem in a static system is not necessarily sufficient to solve the same problem in a dynamic system. Additionally, some existing failure detectors cannot be implemented in dynamic systems. Therefore, it is necessary to redefine existing failure detectors and provide new algorithms.

In this thesis, we provide a new definition of a failure detector for k-set agreement, and prove that it is sufficient to solve k-set agreement in dynamic systems. We also design a dynamic system model and an algorithm that implements this new failure detector.

Additionally, we adapt an existing failure detector for mutual exclusion and prove that it is still the weakest failure detector to solve mutual exclusion in dynamic systems, which means that it is weaker than any other failure detector capable of solving mutual exclusion.

Keywords: distributed systems, failure detectors, dynamic systems, k-set agreement, mutual exclusion

Je dédie cette thèse à Joséphine.

Contents

1 Introduction **1**

1.1 Contributions . . . 3

1.1.1 A Failure Detector for k-Set Agreement in Unknown Dynamic Systems . . 4

1.1.2 A Failure Detector for Mutual Exclusion in Unknown Dynamic Systems . 4

1.1.3 An Asynchronous Reliable Broadcast Algorithm over a Hypercube Topology 4

1.2 Publications . . . 5

1.2.1 Papers in International Conferences . . . 5

1.2.2 Papers in International Journals . . . 5

1.3 Organization of the Manuscript . . . 6

2 A Background on Failure Detectors and Dynamic Systems **7**

2.1 Distributed Systems . . . 8

2.1.1 Processes . . . 8

2.1.2 Communication Models . . . 8

2.1.3 Failure Models . . . 9

2.1.4 Timing Models . . . 9

2.1.5 Notations . . . 9

2.2 Distributed Problems . . . 9

2.2.1 Consensus and k-Set Agreement . . . 10

2.2.2 Fault-Tolerant Mutual Exclusion . . . 11

2.3 Failure Detectors . . . 11

2.3.1 The Failure Detector Hierarchy . . . 12

2.3.2 Failure Detectors for Consensus and k-Set Agreement in Message Passing Systems . . . 12

2.3.3 Failure Detectors for Consensus and k-Set Agreement in Shared Memory Systems . . . 18

2.3.4 Failure Detectors for Mutual Exclusion . . . 19

2.4 Dynamic Networks . . . 19

2.4.1 Increasing and Decreasing Systems with a Static Communication Graph . 20
2.4.2 The Dynamic Graph Model 21
2.4.3 Directed Dynamic Networks 22
2.4.4 Evolving Graphs 22
2.4.5 Time-Varying Graphs (TVG) 23
2.4.6 Unknown Asynchronous Dynamic Networks 25
2.4.7 Summary of Failure Detector Results in Unknown and/or Dynamic Systems 27
2.5 Conclusion 28

3 A Failure Detector for k-Set Agreement in Unknown Dynamic Systems 29
3.1 System Model 30
3.1.1 Process Model 30
3.1.2 Communication Model 31
3.2 Failure Detectors for k-Set Agreement in Unknown Dynamic Systems 32
3.2.1 The $\Sigma_{\perp,k}$ Failure Detector 32
3.2.2 The Family of Failure Detectors $\Pi\Sigma_{\perp,x,y}$ 33
3.3 Assumptions 35
3.3.1 Time-Varying Graph Classes 35
3.3.2 Message Pattern Assumptions 37
3.3.3 Summary of Assumptions 40
3.3.4 Implementation of Message Pattern Assumptions 40
3.3.5 Comparable Assumptions in the Literature 41
3.4 Failure Detector Algorithms 42
3.4.1 An Algorithm for $\Sigma_{\perp,k}$ 42
3.4.2 An Algorithm for $\Pi\Sigma_{\perp,x}$ 46
3.4.3 An Algorithm for $\Pi\Sigma_{\perp,x,y}$ 50
3.5 A k-Set Agreement Algorithm 50
3.5.1 The $Alpha_x$ Sub Protocol 50
3.5.2 $Alpha_x$ Algorithm 51
3.5.3 k-Set Agreement Algorithm 53
3.6 Conclusion 54

4 The Weakest Failure Detector for Mutual Exclusion in Unknown Dynamic Systems 57
4.1 Model and Problem Definition 58
4.1.1 System Model 58
4.1.2 Failure Model 59

4.1.3 Connectivity Model . . . 59
4.1.4 Knowledge Model . . . 60
4.1.5 Problem Definition . . . 60
4.2 Failure Detectors for Mutual Exclusion in Unknown Dynamic Systems . . . 61
4.2.1 The $\mathcal{T}\Sigma^{l}$ Failure Detector . . . 61
4.2.2 The $\mathcal{T}\Sigma^{lr}$ Failure Detector . . . 61
4.3 Sufficiency of $\mathcal{T}\Sigma^{lr}$ to solve Fault-Tolerant Mutual Exclusion . . . 62
4.3.1 Algorithm Description . . . 62
4.3.2 Proof of Correctness . . . 65
4.4 Necessity of $\mathcal{T}\Sigma^{lr}$ to solve Fault-Tolerant Mutual Exclusion . . . 70

5 Conclusion **77**
5.1 Contributions . . . 78
5.1.1 A Failure Detector for k-Set Agreement in Unknown Dynamic Systems . . . 78
5.1.2 A Failure Detector for Recoverable Mutual Exclusion in Unknown Dynamic Systems . . . 78
5.2 Perspectives . . . 79
5.2.1 On the Necessity of Synchronous Processes in Dynamic Systems . . . 79
5.2.2 The Weakest Failure Detector for k-Set Agreement . . . 79
5.2.3 Defining the Mutual Exclusion Problem in Crash-Recovery Systems . . . 79

A A Reliable Broadcast Protocol for Asynchronous Systems with a Hypercube Topology **81**
A.1 Introduction . . . 81
A.2 Related Work . . . 83
A.3 System Model . . . 85
A.4 The VCube . . . 85
A.5 Reliable Broadcast Algorithm for Asynchronous System . . . 86
A.5.1 Message types and local variables . . . 87
A.5.2 Algorithm description . . . 88
A.5.3 Proof of correctness . . . 90
A.6 Performance Discussion . . . 92
A.7 Conclusion and Future Work . . . 93

Bibliography **95**

Chapter 1

Introduction

Contents

1.1 Contributions 3

1.1.1 A Failure Detector for k-Set Agreement in Unknown Dynamic Systems 4

1.1.2 A Failure Detector for Mutual Exclusion in Unknown Dynamic Systems 4

1.1.3 An Asynchronous Reliable Broadcast Algorithm over a Hypercube Topology 4

1.2 Publications 5

1.2.1 Papers in International Conferences 5

1.2.2 Papers in International Journals 5

1.3 Organization of the Manuscript 6

Distributed systems are made of processes that attempt to solve distributed problems by cooperating. These processes can communicate with each other by using shared memory or by exchanging messages. Distributed algorithms describe the steps taken by each process, in particular the read/write operations (in the case of shared memory systems) or the send/receive operations (in the case of message passing systems). The field of distributed computing consists in designing algorithms that solve problems in distributed systems.

A process in a distributed system might fail during the run, and therefore stop following the steps described in its algorithm. A process that fails is called a "faulty" process, while a process that always behaves according to its algorithm is called a "correct" process. Different failure models can be considered: in the crash failure model, a process that fails (crashes) simply stops taking steps for the rest of the run. In the crash/recovery model, a process can crash but might later resume its algorithm. Finally, in the Byzantine model [LSP82], faulty processes can exhibit any arbitrary behaviour, and might even actively work against the algorithm.

A distributed algorithm that solves a problem despite process failures is said to be fault-tolerant.

In message passing systems, some communication models assume that communication is synchronous, which means that there is a bound on message transfer delays and processes are aware of this bound. Other models consider communication to be partially synchronous, which

means that either processes are not aware of the bound on message transfer delays, or the bound is only guaranteed to hold after some unknown time. Finally, some models make no assumption on communication synchrony and consider that there is no bound on message transfer delays: communication is therefore asynchronous.

There are many different models for distributed systems, each making different communication or failure assumptions. A distributed problem can only be solved in certain models. Determining which system assumptions are necessary and/or sufficient to solve a given problem is an important challenge of distributed computing.

A well-known distributed problem is the consensus. In the consensus problem, each process proposes a value, and each correct process must decide a value such that no two different values are decided, and every decided value is a proposed value. The consensus is a fundamental problem in the field of distributed computing. It is used in distributed databases to replicate data while maintaining consistency. A notorious solution to the consensus is the Paxos algorithm [Lam98], which is used, for instance, by Google to maintain consistency between its duplicated indexes.

It was proved in [FLP85] that in the presence of even a single crash failure, it is impossible to solve consensus deterministically in an asynchronous system.

Many articles in the literature have proposed ways to circumvent this impossibility. The partial synchrony model [DLS88] was introduced as a way to detect crashes and solve consensus without the costly assumption of full communication synchrony. The Paxos algorithm [Lam98] tolerates asynchrony while solving a weaker version of consensus where termination is not guaranteed. Another approach is the failure detector abstraction [CT96].

A failure detector is a distributed oracle that provides each process in the system with information, that is not always reliable, on process failures. This information is unreliable in the sense that some correct processes might be falsely suspected of being faulty, while some faulty processes might be believed to be correct. Different classes of failure detectors provide different guarantees on the reliability of the information provided.

A failure detector is said to be weaker than another one if it is possible to implement the first by using the information provided by the second. A failure detector is said to be the weakest to solve a given problem if (1) the information it provides is sufficient to solve the problem and (2) it is weaker than every other detector that solves the problem. By identifying which failure detector class is the weakest to solve a given distributed problem, it is possible to abstract the minimum assumptions needed to solve the problem (such as communication synchrony, or a limitation on the number of failures).

Although many papers in the literature have identified failure detectors sufficient and/or necessary to solve various distributed problems, these results have mostly been confined to static and known systems. However, networks such as wireless, peer-to-peer or ad-hoc networks are inherently dynamic and cannot be modeled as static systems.

In dynamic systems, processes can join or leave the system during the run. Additionally, the communication channels between processes might be intermittent, meaning that the connections between processes can vary during the run, and therefore the communication graph will evolve over time.

In unknown systems, processes do not know the number nor the identities of the participants in the system at the start of the run. Indeed, in a dynamic system where processes might join

the system later in the run, it is unreasonable to expect that every process would know the list of processes from the start.

Unknown dynamic systems pose new challenges that existing algorithms in the literature are not able to deal with. As a result, some existing failure detector definitions cannot be implemented in these new systems, while other failure detectors are no longer sufficient to solve some problems.

Traditionally, failure detectors are used in system models considering static and fully connected communication graphs. These connectivity properties are usually presented as properties of the system model rather than the failure detector augmenting it. When considering a much weaker system model such as a dynamic network, solving any non-trivial problem still requires the assumption of a certain degree of graph connectivity, as not much can be done in a system where no communication link is ever active. Studying dynamic systems means considering the level of temporal connectivity required to solve a specific problem, and using a generic and strong connectivity assumption would defeat that purpose. Instead, the goal should be to use a weak connectivity assumption that is still sufficient to solve the problem. Therefore, to solve a given agreement problem, two things are necessary: (1) a failure detector and (2) connectivity assumptions.

But if connectivity assumptions must be added to the system model in addition to the failure detector, then an argument could be made that the failure detector is not *sufficient* to solve the problem. For this reason, and because in a dynamic system the required level of connectivity is as dependent on the problem as the required failure detector, it makes sense to include connectivity properties in the failure detector definition. Adding these properties should not be seen as strengthening the failure detector, they are still weaker than the assumption of a fully connected, static communication graph.

Another approach is to keep the connectivity properties as part of the system model, separate from the failure detector definition. The advantage of this solution is that the resulting failure detector is only defined in terms of the information it provides on failures. It is therefore closer and more easily comparable to the definitions of existing failure detectors in static systems.

1.1 Contributions

This thesis aims to extend some existing failure detector results, providing new definitions which are more suitable to unknown dynamic systems. Additionally, it provides algorithms that use these new versions of existing failure detectors to solve distributed problems in unknown dynamic systems.

The main contributions of this thesis focus on failure detectors for unknown dynamic systems related to two distributed problems: k-set agreement and mutual exclusion. The first contribution defines a failure detector for k-set agreement that includes connectivity properties, while the second considers connectivity assumptions to be part of the model and defines a failure detector for mutual exclusion strictly in terms of the information it provides on failures.

A third contribution concerns an asynchronous reliable broadcast over an hypercube overlay.

1.1.1 A Failure Detector for k-Set Agreement in Unknown Dynamic Systems

The k-set agreement problem is a generalization of the consensus problem. Every process proposes a value and every correct process must eventually decide a value, such that at most k different values are decided, and every decided value is a proposed value. In static, known systems made of n processes, it has been proved that the $\Pi\Sigma_{x,y}$ failure detector [MRS12] is sufficient to solve k-set agreement in the general case of $1 \leq k < n$ provided that $k \geq xy$, while being the weakest failure detector for the cases of $k = 1$ [CHT96] and $k = n - 1$ [DFGT08].

This thesis provides a new definition of $\Pi\Sigma_{x,y}$ called $\Pi\Sigma_{\perp,x,y}$ [JRAS17]. This new failure detector includes connectivity properties to deal with the dynamics of the system, and uses a default return value $\perp$ to deal with the lack of initial information in an unknown system. An algorithm implementing $\Pi\Sigma_{\perp,x,y}$ in unknown dynamic systems, relying on message pattern assumptions along with connectivity assumptions modeled using the Time-Varying Graph formalism [CFQS12], is presented. Finally, this thesis also provides an algorithm that solves the k-set agreement problem using $\Pi\Sigma_{\perp,x,y}$, with $k \geq xy$, thus proving that $\Pi\Sigma_{\perp,x,y}$ is sufficient to solve k-set agreement in unknown dynamic systems.

1.1.2 A Failure Detector for Mutual Exclusion in Unknown Dynamic Systems

Mutual exclusion is an important distributed problem that is notably used to protect a shared object from being concurrently accessed by multiple processes. In the mutual exclusion problem, processes attempt to access a critical section of their code in such a way that (1) no two processes can be in their critical section at the same time, and (2) whenever some correct process tries to enter its critical section, some correct process must later succeed at entering its critical section. In static, known systems, it has been proved that $(\mathcal{T}, \Sigma^l)$ is the weakest failure to solve mutual exclusion [DGFGK05, BCJ09].

This thesis provides a new definition of $(\mathcal{T}, \Sigma^l)$ called $\mathcal{T}\Sigma^{lr}$ [MJAS19]. This new failure detector is capable of dealing with the lack of information in an unknown system, along with processes that leave the system and rejoin it later (this is modeled as crash/recovery failures). The dynamics of the communication graph are abstracted with the use of connectivity properties defined as part of the system model, and not the failure detector.

An algorithm solving mutual exclusion using $\mathcal{T}\Sigma^{lr}$ is proposed, along with a reduction algorithm showing that any mutual exclusion algorithm can be transformed into $\mathcal{T}\Sigma^{lr}$, thus proving that it is the weakest failure detector to solve mutual exclusion in an unknown dynamic system model.

1.1.3 An Asynchronous Reliable Broadcast Algorithm over a Hypercube Topology

The reliable broadcast abstraction enables each process to reliably communicate information to every other process while preventing situations where only some processes receive the information.

More formally, a reliable broadcast is a communication primitive allowing a process to send a message to all other processes, such that (1) if a correct process broadcasts a message, then it eventually delivers that message; (2) every correct process delivers a message at most once, and

only if that message was previously broadcast by some process; and (3) every correct process delivers the same set of messages [HT93].

The VCube [DBR14] is a distributed overlay that organizes the processes in the system in a virtual hypercube-like topology and includes a diagnosis layer that provides failure detection. A reliable broadcast algorithm leveraging the VCube topology in order to ensure desirable logarithmic properties was provided in [RAJ14], but it requires the system to be synchronous. This thesis introduces a reliable broadcast algorithm for asynchronous systems with a VCube [JRAJ16], that also preserves logarithmic properties depending on the rate of false suspicions.

This work was done as part of the joint project CNRS - Fundação Araucária between the LIP6 (Delys team), the Universidade Federal do Paraná (UFPR) and the Universidade Estadual do Oeste do Paraná (UNIOESTE) in Brazil. Because this contribution does not concern dynamic systems, it will only be presented in the appendix of this thesis.

1.2 Publications

The following articles were published during this thesis.

1.2.1 Papers in International Conferences

- [JRAS15] Denis Jeanneau, Thibault Rieutord, Luciana Arantes, and Pierre Sens. A failure detector for k-set agreement in dynamic systems. In *14th IEEE International Symposium on Network Computing and Applications, NCA 2015, Cambridge, MA, USA, September 28-30, 2015*, pages 176–183, 2015.

- [JRAJ16] Denis Jeanneau, Luiz A. Rodrigues, Luciana Arantes, and Elias Procópio Duarte Jr. An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detector. In *2016 Seventh Latin-American Symposium on Dependable Computing, LADC 2016, Cali, Colombia, October 19-21, 2016*, pages 91–98, 2016.

- [MJAS19] Etienne Mauffret, Denis Jeanneau, Luciana Arantes, and Pierre Sens. The weakest failure detector to solve the mutual exclusion problem in an unknown dynamic environment. In *Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 4-7, 2019 (to be published)*, 2019.

1.2.2 Papers in International Journals

- [JRAS17] Denis Jeanneau, Thibault Rieutord, Luciana Arantes, and Pierre Sens. Solving k-set agreement using failure detectors in unknown dynamic networks. *IEEE Trans. Parallel Distrib. Syst.*, 28(5):1484–1499, 2017.

- [JRAJ17] Denis Jeanneau, Luiz A. Rodrigues, Luciana Arantes, and Elias Procópio Duarte Jr. An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection. *J. Braz. Comp. Soc.*, 23(1):15:1–15:14, 2017.

1.3 Organization of the Manuscript

The rest of this thesis is organized as follows.

Chapter 2 first provides definitions for some common terms, models and problems used throughout this thesis, then summarizes the related work around failure detectors and dynamic systems.

Chapter 3 presents a failure detector for k-set agreement in unknown dynamic systems. It includes the definition of the $\Pi\Sigma_{\perp,x,y}$ failure detector, along with an algorithm implementing the failure detector in an unknown dynamic system and, finally, an algorithm solving k-set agreement using $\Pi\Sigma_{\perp,x,y}$ (provided that $k \leq xy$).

Likewise, Chapter 4 presents a failure detector for mutual exclusion in unknown dynamic systems. It includes the definition of the $\mathcal{T}\Sigma^{lr}$ failure detector, along with an algorithm that solves mutual exclusion using $\mathcal{T}\Sigma^{lr}$ and a reduction algorithm proving that it is the weakest failure detector to solve mutual exclusion in unknown dynamic systems.

Chapter 5 concludes this thesis and proposes some future research directions.

Finally, Appendix A presents an algorithm implementing an asynchronous reliable broadcast over the VCube overlay.

Chapter 2

A Background on Failure Detectors and Dynamic Systems

Contents

2.1 Distributed Systems 8
2.1.1 Processes 8
2.1.2 Communication Models 8
2.1.3 Failure Models 9
2.1.4 Timing Models 9
2.1.5 Notations 9
2.2 Distributed Problems 9
2.2.1 Consensus and k-Set Agreement 10
2.2.2 Fault-Tolerant Mutual Exclusion 11
2.3 Failure Detectors 11
2.3.1 The Failure Detector Hierarchy 12
2.3.2 Failure Detectors for Consensus and k-Set Agreement in Message Passing Systems 12
2.3.3 Failure Detectors for Consensus and k-Set Agreement in Shared Memory Systems 18
2.3.4 Failure Detectors for Mutual Exclusion 19
2.4 Dynamic Networks 19
2.4.1 Increasing and Decreasing Systems with a Static Communication Graph 20
2.4.2 The Dynamic Graph Model 21
2.4.3 Directed Dynamic Networks 22
2.4.4 Evolving Graphs 22
2.4.5 Time-Varying Graphs (TVG) 23
2.4.6 Unknown Asynchronous Dynamic Networks 25
2.4.7 Summary of Failure Detector Results in Unknown and/or Dynamic Systems 27
2.5 Conclusion 28

This thesis focuses on failure detectors for unknown dynamic systems. Thus, it covers two different fields of distributed computing: failure detectors and dynamic systems. In order to provide context, this chapter first summarizes some common definitions used in distributed systems (Section 2.1) along with the definitions of k-set agreement and fault-tolerant mutual exclusion (Section 2.2). Section 2.3 provides a history of failure detectors used to solve consensus, k-set agreement and fault-tolerant mutual exclusion. Finally, Section 2.4 lists different articles attempting to model dynamic distributed systems.

2.1 Distributed Systems

This section describes some existing distributed system models for static and known systems.

2.1.1 Processes

A distributed system is made up of a set of n processes, denoted $\Pi = \{p_1, ..., p_n\}$. Unless specified otherwise, each process knows its own identity, along with the value of n and the set Π.

Each process executes a sequence of steps described by its local algorithm. A step can be the execution of an instruction that only affects the internal state of the process, the sending or reception of a message, or a read/write operation on a shared variable. The set of all local algorithms is called the *distributed algorithm*.

A run is a sequence of steps executed by the processes while respecting the order of local operations as well as the causality of operations (each received message has been previously sent, each value read in a shared variable has been previously written).

2.1.2 Communication Models

There are two main communication models. In the *shared memory* model, processes communicate by reading and writing from/in shared variables. In the *message passing* model, processes communicate by exchanging messages.

The solutions presented in this thesis are based on message passing models. However, because the history of failure detectors in both models is closely intertwined, this chapter also presents failure detectors for shared memory systems.

In a message passing system, processes are connected by *communication links* (or *channels*). Links are often assumed to be bidirectional, although some works consider directed links. A process can only send a message to another process if it is connected to that process by a communication link.

The graph $\mathcal{G} = (V, E)$, where $V = \Pi$ and E is the set of all links in the system, is called the *communication graph* of the system. The majority of the works in the literature make the assumption that the communication graph is connected or complete, meaning that every process can communicate with every other process (through the use of message forwarding in the case of a non-complete but connected graph).

2.1.3 Failure Models

Distributed systems are prone to failures. A process that fails in a run is said to be *faulty* in that run. Conversely, a process that never fails during the run is said to be *correct* in that run. $\mathcal{C}$ denotes the set of all correct processes, while $\mathcal{F}$ is the set of all faulty processes.

The maximum number of faulty processes in a run is denoted f. While the most generic models consider $f \leq n-1$, some models further restrict the number of failures.

The literature considers different types of failures.

In the *crash failure* model, a process that fails stops taking steps for the rest of the run.

In the *crash/recovery failure* model, a process that fails stops taking steps but may recover and resume taking steps later in the run. Processes might have lost part or all of their memory after recovering from a crash: it is therefore important to distinguish between *volatile memory* (which is lost after a crash) and *stable memory* (which is preserved even after a crash).

In the *Byzantine failure* model [LSP82], a process that fails may exhibit arbitrary behaviour, and might even attempt to work against the algorithm.

Similarly to processes, communication links can be subject to failures: messages sent through them can be lost, altered or created. Most papers make the assumptions that links are *reliable* (there is no loss, alteration or creation of messages) or *fair-lossy* (if a same message is sent on a channel infinitely often, the recipient will receive it infinitely often).

2.1.4 Timing Models

Processes can be *synchronous* (there is a bound on the relative speed of processes) or *asynchronous* (there is no such bound).

Similarly, communication can be *synchronous* (there is a known bound on message transfer delays) or *asynchronous* (there is no such bound). Additionally, communication can be *partially synchronous* (the bound on message transfer delays is either unknown, or it only applies after an unknown time) [DLS88].

The system is said to be asynchronous (or *time-free*) if both processes and communication are asynchronous.

Even though model properties are often described with the use of a global clock, the processes themselves do not have access to such a global clock and each process has its own notion of time.

2.1.5 Notations

In this chapter, the $\mathcal{AMP}_{n,f}[A]$ notation is used to designate the asynchronous message passing model comprised of n processes and prone to up to f failures, augmented with assumption A. Similarly, $\mathcal{ASM}_{n,f}[A]$ designates the asynchronous shared memory model. $A = \emptyset$ means that there is no extra assumption. These notations were introduced by Raynal in [Ray10].

2.2 Distributed Problems

Although many distributed problems have been studied in the literature, this thesis focuses on two of them: the k-set agreement problem and the mutual exclusion problem.

2.2.1 Consensus and k-Set Agreement

In the consensus problem, processes attempt to agree on a common value. Each process has access to two functions: *propose*(*value*) and *decide*(*value*). Each process calls the *propose*(*value*) function once to submit a value to the agreement of the other processes. A process calls *decide*(*value*) when it has finally chosen a value. In order to solve the problem, the following properties must be verified:

- **Integrity**: A process decides at most once.
- **Validity**: Any decided value is a proposed value.
- **Agreement**: No two different values are decided.
- **Termination**: Every correct process eventually decides.

The consensus problem is one of the fundamental problems of distributed computing. In many distributed applications, processes have to agree on the state of the application. Such an agreement can be reached with consensus in the following manner: each process proposes its local view of the system to a consensus instance, and the properties of consensus then ensure that all the processes that are alive will eventually adopt the same, consistent view. As a result, consensus is used to implement state machine replication, to provide an atomic broadcast abstraction, or to consistently manage distributed databases.

Fischer, Lynch and Paterson proved in [FLP85] that it is impossible to solve consensus deterministically in an asynchronous system prone to crash failures, even if a single crash failure can occur.

Many solutions have been proposed in the literature to circumvent this impossibility. One approach is to consider a partially synchronous system, which is more realistic than a fully synchronous system while still allowing to solve consensus [DLS88]. Another approach consists in abstracting the timing assumptions made (synchrony or partial synchrony) through the use of failure detectors [CT96]. Finally, one last approach consists in considering a weaker version of the problem: this is the solution chosen for the Paxos algorithm [Lam98], where the termination property is not ensured.

A weaker problem than the consensus is the k-set agreement problem, introduced by Chaudhuri in [Cha93]. It is a generalization of the consensus. Similarly to the consensus, each process has access to the *propose*(*value*) and the *decide*(*value*) functions, and the following properties must be verified:

- **Integrity**: A process decides at most once.
- **Agreement**: At most k different values are decided.
- **Termination**: Every correct process eventually decides.
- **Validity**: Any decided value is a proposed value.

Only the agreement property differs from the consensus. Note that 1-set agreement is equivalent to consensus, and that the greater the value of k, the easier the problem. If the system is made of n processes, then the greatest value of k to be considered is $n-1$, since n-set agreement can be trivially solved without any communication.

A main feature of k-set agreement compared to consensus is that it can be solved in a system composed of k partitions.

In a generalization of the [FLP85] result, it was shown in [BG93] that k-set agreement can only be solved in asynchronous systems if $f < k$, where f is the maximum number of process failures.

2.2.2 Fault-Tolerant Mutual Exclusion

The mutual exclusion problem was introduced by Dijkstra in [Dij65] and has since been widely studied in the literature [Lam74, Lam86, Ray86]. Mutual exclusion algorithms are used by many distributed applications that access shared resources. They must prevent a shared resource from being concurrently accessed by multiple processes.

In the mutual exclusion problem, the code of each process is divided in four sections: remainder section, try section, critical section and exit section. The remainder and critical sections are defined by the higher level application, while the try and exit sections are defined by the mutual exclusion algorithm. A process is said to be well-formed if executes its sections of code in the correct order.

Provided that all processes are well-formed, the fault-tolerant mutual exclusion (FTME) problem is solved if the following properties are verified:

- **Safety**: Two distinct processes cannot be in their critical section at the same time.
- **Liveness**: If a correct process enters its try section, then at some time later some correct process enters the critical section. Additionally, if a correct process is in its exit section, at some time later it enters it remainder section.

The following additional fairness property is often considered along with FTME:

- **Starvation Freedom**: If no process stays forever in its critical section, then every correct process that reaches its try section eventually enters its critical section.

It was proved in [DGFGK05] that, if a majority of processes are correct, then any algorithm solving FTME can be transformed into an algorithm that also verifies the starvation freedom property.

Several mutual exclusion algorithms that tolerate crash failures have been proposed in the literature [NLM90, AEA91, SAS06]. Furthermore, mutual exclusion algorithms that tolerate crash-recovery failures in the shared memory model were proposed in [GR16, GH17, JJ17], with shared variables being stored in stable memory. One crash-recovery mutual exclusion algorithm for message passing systems was proposed in [CSL90], relying on the assumption that failures do not occur in adjacent connected processes.

2.3 Failure Detectors

The failure detector abstraction has been investigated as a way to circumvent the impossibility result of [FLP85] and solve consensus in asynchronous systems prone to crash failures [CHT96].

Failure detectors [CT96] are distributed oracles that provide processes with information on process failures, often either as a list of suspected process identities, or as a list of trusted ones. This information is unreliable in the sense that the failure detector may erroneously consider a correct process as faulty, or vice versa, but will attempt to correct these mistakes later. Each failure detector class ensures some properties on the reliability of the failure information. A failure detector is an abstraction of the system assumptions that are necessary and/or sufficient to solve a given problem.

A process p is said to *suspect* another process q if p has detected q as faulty. Conversely, p is said to *trust* q if p has not detected q as faulty.

In [CT96], Chandra and Toueg define failure detector classes using completeness and accuracy properties:

- **Strong Completeness**: Eventually, every process that crashes is suspected by *every* correct process.
- **Weak Completeness**: Eventually, every process that crashes is suspected by *some* correct process.
- **Strong Accuracy**: No process is suspected before it crashes.
- **Weak Accuracy**: Some correct process is never suspected.

The accuracy properties can be declined into the *eventually strong accuracy* and *eventually weak accuracy*, where the property only applies after some unknown time.

Using these properties, Chandra and Toueg defined eight classes of failure detectors: $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, $\diamond\mathcal{S}$, $\mathcal{Q}$, $\mathcal{W}$, $\diamond\mathcal{Q}$ and $\diamond\mathcal{W}$.

The notation $\mathcal{D}_p(t)$ is used to designate the output of failure detector $\mathcal{D}$ at time t on process p.

2.3.1 The Failure Detector Hierarchy

Failure detectors can be partially ordered based on their "power". A stronger failure detector can be used to solve more difficult problems.

A failure detector $\mathcal{D}_1$ is said to be *weaker* than $\mathcal{D}_2$ if there exists a distributed algorithm that can implement $\mathcal{D}_1$ using the information on failures provided by $\mathcal{D}_2$. Intuitively, this means that the computing power provided to the system by $\mathcal{D}_2$ is stronger than the computing power provided by $\mathcal{D}_1$. A failure detector that is sufficient to solve a given problem while being weaker than every other failure detector that can solve it, is said to be the *weakest failure detector* to solve that problem. It follows that the weakest failure detector to solve a problem can be implemented in any system in which the problem can be solved.

2.3.2 Failure Detectors for Consensus and k-Set Agreement in Message Passing Systems

While the weakest failure detectors to solve consensus and set agreement have been identified, the generic case of k-set agreement for $1 \leq k \leq n-1$ in $\mathcal{AMP}_{n,f}[\emptyset]$ seems to be a harder problem, as to this day the weakest failure detector to solve it has not yet been identified. Although most of the results take the form of sufficient failure detectors, [BRS11] proved that the assumption that $k > \frac{n-1}{n-f}$ is necessary to solve k-set agreement in message passing systems.

Table 2.1 (along with Table 2.2 in the next section) is an updated extension of the table presented by Raynal in [Ray11] and summarizes the failure detectors used to solve consensus and k-set agreement in the literature.

The "Failure pattern" column in Table 2.1 indicates the additional assumption on the number of failures necessary for the failure detector to ensure the property of column "Property".

FD class	Article	Section	Property	Failure pattern
$\diamond\mathcal{S}$, Ω	[CHT96]	2.3.2.1	Weakest for consensus	$f < n/2$
(Ω_k, Σ_k)	[BR09]	2.3.2.2	Insufficient for k-set agreement	$f \leq n-1$
$\mathcal{S}_x$	[MR99]	2.3.2.3	Solves k-set agreement	$f < k+x-1$
$\diamond\mathcal{S}_x$	[MR99]	2.3.2.3	Solves k-set agreement	See 2.3.2.3
Σ	[DFG10]	2.3.2.4	Weakest for registers	$f \leq n-1$
(Ω, Σ)	[DFG10]	2.3.2.4	Weakest for consensus	$f \leq n-1$
Σ_k	[BR09]	2.3.2.5	Necessary for k-set agreement	$f \leq n-1$
Π_k (2009)	[BR09]	2.3.2.5	Same power as (Ω_k, Σ_k)	$f \leq n-1$
$\mathcal{L}$	[DFGT08]	2.3.2.6	Weakest for $(n-1)$-set agreement	$f \leq n-1$
$\mathcal{L}_k$	[BRS09]	2.3.2.7	Solves k-set agreement with $k \geq n/2$	$f \leq n-1$
$\Pi\Sigma_{x,y}$	[MRS12]	2.3.2.8	Solves k-set agreement, weaker than $(\overline{\Omega}_x, \Sigma_y)$	$f \leq n-1$
Π_k(2007)	[CZCL07]	2.3.2.9	Solves k-set agreement, weaker than (Ω_k, Σ)	$f \leq n-1$
Π_k^s	[CZCL07]	2.3.2.9	Solves k-set agreement, weaker than Π_k (2007)	$f \leq n-1$
$(\diamond\mathcal{S}_{bl}, \Sigma_{bl})$	[FMR05]	2.3.2.10	Same power as (Ω, Σ)	$f \leq n-1$

Table 2.1: A global view of failure detectors solving agreement problems in message passing systems

2.3.2.1 The $\diamond\mathcal{S}$ Eventually Strong Failure Detector and the Ω Eventual Leader Detector

$\diamond\mathcal{S}$ was introduced by Chandra and Toueg in [CT96]. It provides processes with a list of processes suspected of being faulty, such that:

- **Strong completeness**: $\forall p \in \mathcal{F}, \exists \tau \in \mathcal{T}, \forall t > \tau, \forall q \in \mathcal{C} : p \in \diamond\mathcal{S}_q(t)$
- **Eventual weak accuracy**: $\exists l \in \mathcal{C}, \exists \tau \in \mathcal{T}, \forall t > \tau, \forall p \in \mathcal{C} : l \notin \diamond\mathcal{S}_p(t)$

Strong completeness means that eventually, all faulty processes are suspected by all correct processes. Eventual weak accuracy means that there is at least one correct process which is eventually not suspected by any correct process.

The eventual leader detector Ω provides each process with a process identity such that the following property is ensured:

- **Eventual leadership**: $\exists l \in \mathcal{C}, \exists \tau \in \mathcal{T}, \forall p \in \mathcal{C}, \forall t \geq \tau : \Omega_p(t) = l$

Eventual leadership means that after a certain time τ, all processes will trust the same correct process. The detector eventually provides the processes with a correct leader, but there is no way to know τ.

Ω was proven in [CHT96] to be the weakest failure detector with which to enrich $\mathcal{ASM}_{n,n-1}[\emptyset]$ in order to solve consensus. Since Attiya et al. proved in [ABD95] that read/write registers can be simulated in a message passing system if and only if $f < n/2$, it follows that Ω is also the weakest failure detector to solve consensus in $\mathcal{AMP}_{n,f}[f < n/2]$.

The Ω failure detector is equivalent to $\diamond\mathcal{S}$, provided that the membership of the system is known [JAF06].

2.3.2.2 The Ω_k Eventual Leader Detector Family

Although Ω is minimal for the case $k = 1$ in $\mathcal{ASM}_{n,n-1}[\emptyset]$ and $\mathcal{AMP}_{n,f}[f < n/2]$, it is too strong for k-set agreement with $1 < k \leq n - 1$. The eventual leader detector family Ω_k was introduced in [Nei95] as a generalization of Ω aimed to solve k-set agreement. It provides processes with a set of process identities such that:

- **Eventual leadership**: $\exists L \subseteq \Pi, |L| \leq k, L \cap \mathcal{C} \neq \emptyset, \exists \tau \in \mathcal{T}, \forall p \in \mathcal{C}, \forall t \geq \tau : \Omega_{kp}(t) = L$

The eventual leadership property implies that every correct process will ultimately trust the same process group, in which there is at least one correct process. Note that Ω_1 is equivalent to Ω. In [RT06], the author conjectured that Ω_k was the weakest failure detector to solve k-set agreement in the shared memory model, but this conjecture was disproved in [CZCL07].

2.3.2.3 The $\mathcal{S}_x$ and $\diamond\mathcal{S}_x$ Limited Scope Accuracy Failure Detectors

Failure detectors with limited scope accuracy were introduced in [MR99], denoted $\mathcal{S}_k$ and $\diamond\mathcal{S}_k$. Since the k parameter is unrelated to the k-set agreement problem, in this chapter they will be denoted $\mathcal{S}_x$ and $\diamond\mathcal{S}_x$. They are defined relatively to the Strong failure detector $\mathcal{S}$ and the eventually Strong failure detector $\diamond\mathcal{S}$, respectively. $\mathcal{S}_x$ relies on the following properties:

- **Strong completeness**: $\forall p \in \mathcal{F}, \exists \tau \in \mathcal{T}, \forall t > \tau, \forall q \in \mathcal{C} : p \in \mathcal{S}_{x,q}(t)$
- **x-Accuracy**: $\exists Q \subseteq \Pi, |Q| = x, Q \cap \mathcal{C} \neq \emptyset, \exists p \in Q \cap \mathcal{C} : \forall q \in Q, \forall t \in \mathcal{T} : p \notin \mathcal{S}_{x,q}(t)$.

The strong completeness property is the same as the one of $\mathcal{S}$ and $\diamond\mathcal{S}$: eventually, all faulty processes are suspected by all correct processes. The x-accuracy property means that there is one correct process that is not suspected by a set of x processes (including itself). $\mathcal{S}_x$ ensures

the strong completeness and the x-accuracy properties. $\diamond\mathcal{S}_x$ ensures the strong completeness and eventually ensures the x-accuracy property.

[MR99] provides an algorithm extracting $\mathcal{S}$ from $\mathcal{S}_x$ and $\diamond\mathcal{S}$ from $\diamond\mathcal{S}_x$, if $f < x$. It also presents an algorithm solving consensus with $\diamond\mathcal{S}_x$, provided that $f < min(x, n/2)$.

Algorithms for k-set agreement with this family of failure detectors can be found in [MR00]: a first algorithm requires $\mathcal{S}_x$ and $f < k+x-1$. Another algorithm makes use of $\diamond\mathcal{S}_x$ and requires $f < max(k, max_{1\leq\alpha\leq k}(min(n - \alpha\lfloor n/(\alpha+1)\rfloor, \alpha + x - 1)))$.

2.3.2.4 The Σ Quorum Detector

The quorum detector Σ, introduced in [DFG10] by Delporte-Gallet, Fauconnier and Guerraoui provides each process p_i with a quorum of process identities Σ_{p_i} such that the following properties hold:

- **Intersection**: $\forall t_1, t_2 \in \mathcal{T}, \forall p_1, p_2 \in \Pi : \Sigma_{p_1}(t_1) \cap \Sigma_{p_2}(t_2) \neq \emptyset$
- **Completeness**: $\exists \tau \in \mathcal{T}, \forall p \in \mathcal{C}, \forall t \in \mathcal{T}, t \geq \tau : \Sigma_p(t) \subseteq \mathcal{C}$

Intersection ensures that all the quorums formed by any two process at any two distinct times intersect. Completeness guarantees that at some point in time, all the quorums will only contain correct processes. [DFG10] proves that Σ is the weakest failure detector to implement read/write registers in $\mathcal{AMP}_{n,n-1}[\emptyset]$. Such a result also implies that the pair (Ω, Σ) is the weakest failure detector to solve consensus in $\mathcal{AMP}_{n,n-1}[\emptyset]$. Contrarily to Ω (or $\diamond\mathcal{S}$), it does not require a majority of correct processes. It is, however, too strong for k-set agreement with $k > 1$.

2.3.2.5 The Σ_k Quorum Detector Family

The quorum detector family Σ_k was introduced by Bonnet and Raynal in [BR09]. Similarly to Σ, it provides each process with a list of process identities such that the following properties are satisfied:

- **Intersection**: $\forall t_1, ..., t_{k+1} \in \mathcal{T}, \forall p_1, ..., p_{k+1} \in \Pi, \exists i, j : 1 \leq i \neq j \leq k+1 : \Sigma_{kp_i}(t_i) \cap \Sigma_{kp_j}(t_j) \neq \emptyset$
- **Completeness**: $\exists \tau \in \mathcal{T}, \forall p \in \mathcal{C}, \forall t \in \mathcal{T}, t \geq \tau : \Sigma_{kp}(t) \subseteq \mathcal{C}$

The intersection property ensures that out of any $k+1$ quorums formed on any processes at any time, at least two will intersect. The completeness ensures that eventually quorums contain only correct processes. Intuitively, Σ_k prevents the network from partitioning into more than k independent subsets.

[BR09] also proved that Σ_k is necessary for k-set agreement and introduced Π_k (unrelated to the Π_k of [CZCL07]), a class of failure detectors proved to have the same computational power as (Ω_k, Σ_k).

Given that (Ω, Σ) is the weakest failure detector to solve consensus in $\mathcal{AMP}_{n,n-1}[\emptyset]$, (Ω_k, Σ_k) was conjectured in [BR09] to be the weakest failure detector to solve k-set agreement in the message passing model, but it was proved that (Ω_k, Σ_k) is not sufficient to solve it with $1 < k < n-1$ [BT10, BRS11]. More specifically, [BT10] proves that (Ω_x, Σ_z) cannot solve k-set agreement if $k \geq xz$. Later, [MRS12] also proved that Ω_k is not necessary for k-set with $1 < k < n-1$.

2.3.2.6 The $\mathcal{L}$ Loneliness Detector

As a first step to find tight bounds on generic k-set resolvability for $1 \leq k \leq n-1$, researchers started by working on the other extreme case: $(n-1)$-set agreement, also called set agreement. Although Σ is not sufficient to solve consensus and was conjectured by Zielinski in [Zie08] to be the weakest failure detector to solve set-agreement, it was proved in [DFG08] that it is too strong for set agreement in message passing systems. Therefore, a weaker detector than Σ is required.

The loneliness detector $\mathcal{L}$ was first introduced in [DFGT08]. It is a very weak failure detector, as it only provides processes with the information that they are alone. Intuitively, $\mathcal{L}$ detects the situation where a process is the only remaining correct process in the system. More formally, $\mathcal{L}$ provides each process with a boolean variable such that:

- **Stability**: $\exists p \in \Pi, \forall t \in \mathcal{T} : \mathcal{L}_p(t) = false$
- **Loneliness**: $\forall p \in \Pi, (\mathcal{C} = \{p\}) \Rightarrow \exists \tau \in \mathcal{T}, \forall t \in \mathcal{T}, t \geq \tau : \mathcal{L}_p(t) = true$

The stability property ensures that there is at least one process that never considers itself alone. The loneliness property guarantees that if a correct process is alone, it will eventually know that it is alone. [DFGT08] proved that $\mathcal{L}$ is the weakest failure detector to solve set agreement in $\mathcal{AMP}_{n,n-1}[\emptyset]$.

2.3.2.7 The $\mathcal{L}_k$ Generalized Loneliness Detector

$\mathcal{L}_k$ is introduced in [BRS09, BRS14] as a generalization of the loneliness detector $\mathcal{L}$ ($\mathcal{L}$ is $\mathcal{L}_{n-1}$) to detect $(n-k)$-loneliness. Similarly to $\mathcal{L}$, it outputs to each process a boolean value such that the following properties hold:

- **Stability**: $\exists \Pi_0 \subseteq \Pi, |\Pi_0| = n-k, \forall p \in \Pi_0, \forall t \in \mathcal{T} : \mathcal{L}_{kp}(t) = false$
- **Loneliness**: $(f \geq k) \Rightarrow \exists \tau \in \mathcal{T}, \exists p \in \mathcal{C}, \forall t \geq \tau : \mathcal{L}_{kp}(t) = true$

The stability property imposes that there is a set of $n-k$ processes that always know that they are not alone. The loneliness property ensures that if there are fewer than $n-k$ correct processes, then at least one correct process returns true.

$\mathcal{L}_k$ solves k-set agreement but is not minimal. As shown in [MRS11], $\mathcal{L}_k$ is only *realistic* (meaning it can be implemented in a synchronous system with $f = n-1$) if and only if $k \geq n/2$.

The same paper introduced $\diamond\mathcal{L}_k$, the eventual loneliness detector, and showed that it is equivalent to Ω_k.

2.3.2.8 The $\Pi\Sigma_{x,y}$ Failure Detector

This failure detector was introduced by Mostéfaoui, Raynal and Stainer in [MRS12] and defined incrementally. The paper starts by defining the intermediary failure detector $\Pi\Sigma_x$, which provides each process with a quorum following the properties of Σ_k, plus a variable *leader* providing the following additional property:

- **Eventual partial leadership**:

$$\exists l \in \mathcal{C}, \forall p \in \mathcal{C} : (\forall \tau \in \mathcal{T}, \exists \tau_p, \tau_l \geq \tau : \Pi\Sigma_{x,p}(\tau_p) \cap \Pi\Sigma_{x,l}(\tau_l) \neq \emptyset)$$

$$\Rightarrow (\exists \tau \in \mathcal{T}, \forall t \geq \tau : leader_p(t) = l)$$

The eventual partial leadership property means that there is a process l such that, for every process p whose quorum intersects infinitely often with l's quorum, eventually l is forever the leader of p.

$\Pi\Sigma_{x,y}$ can be seen as a set of y detectors of class $\Pi\Sigma_x$ of which only one has to follow the eventual leadership property. It is weaker than $(\Sigma_x, \overline{\Omega}_y)$ and solves k-set agreement in the message passing model for $k \geq xy$ (which is a necessity proved in [BT10]).

The authors in [MRS12] provide an intuitive description of $\Pi\Sigma_{x,y}$. $\Pi\Sigma_{x,1}$ (1) prevents the system from partitioning into more than x partitions with the properties of Σ_x and (2) guarantees that the processes of at least one of these subsets agree on a common leader. $\Pi\Sigma_{x,y}$ can be seen as y independent instances of $\Pi\Sigma_x$ in which the guarantee (2) has to be hold in only one of these instances.

2.3.2.9 The Π_k and Π_k^s Partitioned Failure Detectors

The partitioned failure detectors are a set of failure detectors introduced in [CZCL07]. The goal is to weaken existing failure detectors by separating their safety and liveness properties. While it is necessary to always ensure the safety property for all processes, the liveness property is only required to be ensured for one component of the graph. Based on this approach, the authors deduce several new failure detectors and prove that they are weaker than the existing ones.

Π_k is the partitioned variant of (Ω_k, Σ). Π_k^s is a weaker version of Π_k using dynamic partitioning during the run. An algorithm solving k-set agreement in $\mathcal{AMP}_{n,f}[\Pi_k^s]$ and inspired from the Paxos algorithm of [Lam98] is provided in [CZCL07].

2.3.2.10 The $\diamond\mathcal{S}_{bl}$ and Σ_{bl} Bounded Lifetime Failure Detectors

The bounded lifetime failure detectors were introduced by Friedman et al. in [FMR05]. These failure detectors provide the upper layer application with a $fd\_end()$ primitive, which can be invoked by the latter to inform the failure detector that it no longer requires the failure detector output. As a result, some failure detector properties are not required to be verified all the time.

$\diamond\mathcal{S}_{bl}$ is the bound lifetime variant of $\diamond\mathcal{S}$. Similarly to the original, it outputs a list of suspected processes, and must verify the same strong completeness property. Additionally, the following property must be verified:

- **Bounded lifetime eventual weak accuracy**: There is a time t and a time t_e such that from t until t_e, there is a process that is alive at t_e and that is not suspected by all the processes that have not crashed at time t_e.

t_e is the time at which $fd\_end()$ is invoked. After t_e, the accuracy property no longer applies. In the article, the authors show that an existing $\diamond\mathcal{S}$-based consensus algorithm can be adapted to use $\diamond\mathcal{S}_{bl}$. The algorithm is modified so that every process invokes $fd\_end()$ after it decides. Such an algorithm still solves consensus, while allowing the failure detector to only ensure the accuracy property for some time.

The Σ_{bl} failure detector is also introduced, in the same article, as the bounded lifetime variant of Σ. The local output history of Σ_{bl} for each process is delimited in time intervals, such that

a new interval starts every time $fd\_end()$ is invoked. The intersection property only applies to pairs of quorums that were formed during consecutive or concurrent time intervals.

The paper adapts an existing Σ based atomic register protocol for Σ_{bl} by invoking $fd\_end()$ at the end of every read or write operation.

2.3.3 Failure Detectors for Consensus and k-Set Agreement in Shared Memory Systems

FD class	Article	Section	Property
$\diamond\mathcal{S}$, Ω	[CT96, CHT96]	2.3.2.1	Weakest for consensus
Ω_k	[Nei95]	2.3.2.2	Solves k-set agreement
$\diamond\mathcal{L}_k$	[MRS11]	2.3.2.7	Equivalent to Ω_k
$\Pi\Omega_k$	[CZCL07]	2.3.3.1	Solves k-set agreement, weaker than Ω_k
$\Pi\Omega\Upsilon_k$	[CZCL07]	2.3.3.1	Solves $(n-1)$-set agreement, weaker than Υ
$\overline{\Omega}$	[Zie08]	2.3.3.2	Weakest for $(n-1)$-set agreement
Υ	[GHK+07]	2.3.3.2	Sufficient for $(n-1)$-set agreement
$\overline{\Omega}_k$	[Ray07]	2.3.3.3	Weakest for k-set agreement

Table 2.2: A global view of failure detectors solving agreement problems in shared memory systems

2.3.3.1 The $\Pi\Omega_k$ and $\Pi\Omega\Upsilon_k$ Partitioned Failure Detectors

$\Pi\Omega_k$ and $\Pi\Omega\Upsilon_k$ are two of the partitioned failure detectors presented in [CZCL07] (see Section 2.3.2.9).

$\Pi\Omega_k$ is the partitioned variant of Ω_k. The article provides an algorithm solving k-set agreement in $\mathcal{ASM}_{n,n-1}[\Pi\Omega_k]$ and proves that it is weaker than Ω_k, thus disproving the conjecture in [RT06] that Ω_k is the weakest failure detector to solve k-set agreement in shared memory systems.

$\Pi\Omega\Upsilon_k$ is a partitioned failure detector combining features of $\Pi\Omega_k$ and Υ, while being weaker than both of them. It solves $(n-1)$-set agreement in the shared memory model.

2.3.3.2 The $\overline{\Omega}$ Anti-Omega Failure Detector

Inspired by the Υ detector introduced in [GHK+07], Zielinski introduced in [Zie08] the $\overline{\Omega}$ failure detector that solves the set agreement problem in shared memory systems.

Υ provides each process with a non-empty set of process identities which is eventually ensured to be the same for every process and different from $\mathcal{C}$.

$\overline{\Omega}$ is strictly weaker than Υ and provides each process with a process identity, with only the guarantee that there is one correct process whose id will be output a finite number of times. [Zie08] proves that $\overline{\Omega}$ is the weakest failure detector to solve set agreement in $\mathcal{ASM}_{n,n-1}[\emptyset]$.

2.3.3.3 The $\overline{\Omega}_k$ Failure Detector

$\overline{\Omega}_k$ is a generalization of $\overline{\Omega}$ such that $\overline{\Omega}$ is $\overline{\Omega}_{n-1}$ and Ω is $\overline{\Omega}_1$. It was introduced in [Ray07] and conjectured to be the weakest failure detector for k-set agreement in the shared memory model. This conjecture was later proved to be true in [GK09]. $\overline{\Omega}_k$ provides each process with a list of process identities such that:

- **Validity**: $\forall p \in \Pi, \forall t \in \mathcal{T}, \overline{\Omega}_{kp}(t) \subseteq \Pi : |\overline{\Omega}_{kp}(t)| = k$
- **Weak eventual leadership**: $\exists \tau \in \mathcal{T}, \exists l \in \mathcal{C}, \forall t \in \mathcal{T}, t \geq \tau, \forall p \in \Pi : l \in \overline{\Omega}_{kp}(t)$

Weak eventual leadership implies that there is a correct process l that will ultimately be in the set output by $\overline{\Omega}_k$ for every process. This characteristic makes $\overline{\Omega}_k$ slightly different from Ω_k.

2.3.4 Failure Detectors for Mutual Exclusion

This section presents the failure detectors used to solve the FTME problem in message passing systems.

2.3.4.1 The Trusting Failure Detector

In [DGFGK05], Delporte-Gallet et al. introduced the trusting failure detector $\mathcal{T}$ and proved that it is the weakest failure detector with which to enrich $\mathcal{AMP}_{n,f}[f < n/2]$ to solve FTME. $\mathcal{T}$ provides each process p with a list of trusted processes. The following properties must be verified.

- **Eventually strong accuracy**: Every correct process p is eventually trusted forever by every correct process.
- **Strong completeness**: Every faulty process p is eventually suspected forever by every correct process.
- **Trusting accuracy**: For any processes p and q, if there exist times t and $t' > t$ such that p trusts q at time t and p suspects q at time t', then q is faulty.

2.3.4.2 The Σ^l Failure Detector

Bhatt et al. introduce in [BCJ09] the Σ^l quorum failure detector. Σ^l is a variant of the Σ quorum failure detector adapted for the FTME problem. It provides each process p with a quorum of process identities. Σ^l verifies the following properties.

- **Strong completeness**: Every faulty process p is eventually suspected forever by every correct process.
- **Live pairs intersection**: If two processes p and q are both alive at time t, then for any couple of time instants $t_1 \leq t$ and $t_2 \leq t$, $\Sigma^l_p(t_1) \cap \Sigma^l_q(t_2) \neq \emptyset$.

Bhatt et al. show in [BCJ09] that $\mathcal{T}$ and Σ^l used together, denoted $(\mathcal{T}, \Sigma^l)$, constitute the weakest failure detector with which to enrich $\mathcal{AMP}_{n,n-1}[\emptyset]$ to solve FTME.

2.4 Dynamic Networks

All the articles discussed in the previous sections made the assumption that the distributed system is composed of a finite set of processes, that the membership of the system only changes due to failures, and that the communication graph is static during the run. These assumptions do not apply to dynamic systems.

There is no consensual formal definition of a dynamic network. Some papers define a dynamic system as a distributed system in which the communication graph evolves over time, while others define it as a model where processes can join and leave the system during a run. Some articles mix both of those proposals in the same model. A few articles have attempted to regroup other authors' proposals in order to formalize common models [BF03, Agu04, KLO10, CFQS12, BRS12].

2.4.1 Increasing and Decreasing Systems with a Static Communication Graph

In increasing and decreasing systems, processes can join or leave the system during the run, but the communication graph is static over time. This section regroups a number of papers which consider such systems.

2.4.1.1 Infinitely Many Processes.

In [Agu04], Aguilera defines models where processes can join or leave the system during the run:

- In the **finite arrival model**, the system has infinitely many processes, but each run has only finitely many. As a result, algorithms do not know the bound on the number of processes.
- In the **infinite arrival model**, the system has infinitely many processes and each run can have infinitely many processes, but in each finite time interval, only finitely many processes take steps.
- In the **infinite concurrency model**, the system has infinitely many processes, each run can have infinitely many processes and each finite time interval can have infinitely many processes.

A number of papers have explored these models [MRT+05, CRTW07, BBRT07, AGSS13], sometimes combining them with a communication model addressing graph dynamics (e.g. TVG, see section 2.4.5).

Mostéfaoui et al. make use in [MRT+05] of the *query-response* mechanism, in which processes broadcast a query to their neighbours and wait for a certain number of responses. Each process waits for α responses to its query, where α is the number of processes eventually guaranteed to respond. With this mechanism, processes are not required to know the number of processes in the system, which allows for a finite arrival model. The paper provides a Ω algorithm for an asynchronous finite arrival model using query-response.

[CRTW07] uses a model composed of two sets of nodes: fixed support stations forming a static complete graph, asynchronous but entirely known, and mobile hosts which follow the

finite arrival model and attach themselves to support stations. Using this model, the paper provides an algorithm for Ω using message pattern assumptions.

In [ADGF08], Abboud et al. consider that processes cannot crash, but can join the system during the run and those that did not join the system yet at a given time are considered faulty at that time. The paper shows that consensus cannot be solved in this model but can be solved provided that the model is enriched with Σ. The authors remark that Σ can be implemented with an estimation m of the number of processes such that $\lfloor n/2 \rfloor + 1 \leq m \leq n$.

2.4.1.2 The Dynamic Eventual Leader Failure Detector $\Delta\Omega$.

[LRAC12] introduces $\Delta\Omega$ as an eventual leader definition adapted for dynamic networks. It is defined by the following properties, where $\Pi(t)$ is the system membership at time $t \in \mathcal{T}$ and $join_i$ is the time at which process p_i joined the system:

- **Eventual Leadership in Non-Increasing systems**:

$$[\exists \tau \in \mathcal{T} : \forall \tau_1, \tau_2 \geq \tau : (\tau_1 \geq \tau_2) \Rightarrow (\Pi(\tau_1) \subseteq \Pi(\tau_2))]$$

$$\Rightarrow [\exists p_l \in \bigcap_{\tau' \geq \tau} \Pi(\tau') : \exists \tau_1 \geq \tau : \forall \tau_2 \geq \tau_1 : \forall p_i \in \Pi(\tau_2) : \Omega_i(\tau_2) = l]$$

- **Eventual Leadership in Non-Decreasing systems**:

$$[\exists \tau \in \mathcal{T} : \forall \tau_1, \tau_2 \geq \tau : (\tau_1 \leq \tau_2) \Rightarrow (\Pi(\tau_1) \subseteq \Pi(\tau_2))] \Rightarrow [\exists \tau_1 \geq \tau, \exists p_l \in \Pi(\tau_1) :$$

$$[\forall \tau_2 \geq \tau_1 : \forall p_i \in \Pi(\tau) : \Omega_i(\tau_2) = l] \wedge$$

$$[\forall \tau' \geq \tau : (p_i \in \Pi(\tau') \backslash \Pi(\tau)) \Rightarrow (\exists \tau_0 \geq join_i : \forall \tau'' \geq \tau_0 : \Omega_i(\tau'') = l)]]$$

Intuitively, the first property ensures that if after some time no new process joins the system (finite arrival), then a leader is eventually elected. The second property ensures that if after some time no process leaves the system, then a leader is eventually elected and new processes eventually adopt it.

The paper provides two algorithms implementing $\Delta\Omega$: the first one assumes that processes synchronize their local clocks with others when first entering the system, the second one assumes reliable channels. An extension of $\Delta\Omega$ accounting for an evolving communication graph is presented in [GLLR13]. It uses exclusively assumptions on "good periods", and includes a survey and comparison of different models and algorithms, mostly with partial synchrony assumptions.

2.4.2 The Dynamic Graph Model

Kuhn, Lynch and Oshman present in [KLO10] the Dynamic Graph Model that considers a fixed set of nodes although the authors remark that this is not a fundamental characteristic of the model, which could be extended for a dynamic set of participants. Algorithms in this model work in synchronous rounds (i.e., communication links are synchronous) and make use of broadcasts as a communication primitive. Nodes are assumed reliable (crash failures are not considered). The dynamics of the system lie in the communication graph which changes in each

round. Connectivity assumptions are expressed around the concept of *T-interval connectivity*, which imposes the existence of a stable connected subgraph for every T consecutive rounds. The stable subgraph is not known to the nodes in advance. In every round the active communication links are chosen by an adversary which abstracts above assumptions. Other than the model itself, the paper presents solutions to several distributed problems such as *counting* and *k-token dissemination*, but not consensus.

In [KMO11], the authors extend the model and propose algorithms for consensus, *simultaneous consensus* (all the nodes decide in the same round) and Δ-*coordinated consensus* (all the correct processes decide within Δ rounds of each other).

[SS14] studies the problem of k-set agreement in the Dynamic Graph Model of [KLO10] while considering $p - partitioned\ networks$ (at each round, the communication graph consists of at most p connected components). The paper proves the necessity of knowing at least an upper bound on the number of processes if the graph is at least 2-partitioned, and provides an algorithm solving k-set agreement under that condition.

2.4.3 Directed Dynamic Networks

In [BRS12], Biely, Robinson and Schmid present a dynamic model close but weaker than the Dynamic Graph Model. This new model also revolves around synchronous rounds, but considers directed communication graphs that do not need to be connected at every round (therefore excluding the concept of *T-interval connectivity*) and uses the weaker assumption of *vertex-stable root components* instead. In every round, there must be exactly one strongly connected component that has only out-going links to some of the remaining processes and can reach every process in the system. The paper presents an algorithm for consensus under this assumption.

An algorithm for binary consensus (consensus with binary values) is provided in [CG13] in a similar model with directed communication graphs. The paper proposes an algorithm for consensus with a connectivity assumption which is proven necessary: there exists a same process that has a directed path towards all other processes for every round of the run.

In [BRS$^+$18] Biely et al. introduce the concept of gracefully degrading consensus: instead of solving k-set agreement for a preset value of k, the algorithm adapts to the network conditions it encounters during the run and attempts to provide agreement with the lowest possible value of k. To this end, the authors present both an algorithm and its proof using the same model as [BRS12] with similar assumptions on *vertex-stable root components*. The article also gives minimality results regarding temporal stability and information flow.

2.4.4 Evolving Graphs

The Evolving Graphs model was introduced by Bhadra and Ferreira in [BF03]. An evolving graph is defined as a system $\mathcal{G} = (G, \mathcal{S}_G)$ where $G(V, E)$ is a digraph and $\mathcal{S}_G = G_0, G_1, ...G_\tau, \tau \in \mathcal{T}$ is an ordered sequence of its subgraphs $G_i(V_i, E_i)$. This formalism enables nodes to join or leave the system, as well as a dynamic communication graph.

If P is a path in G, then let $F(P)$ be its source and $L(P)$ its destination. A path in $\mathcal{G}$ is a sequence $P_\mathcal{G}(u, v) = P_{t_1}, P_{t_2}, ..., p_{t_x}$ with $t_1 < t_2 < ... < t_x$, such that P_{t_i} is a defined path in G_{t_i} and $F(P_{t_1}) = u, L(P_{t_x}) = v$, and $\forall i < x : L(P_{t_i}) = F(P_{t_{i+1}})$.

This definition is extended in [BF03, BFJ03] with the concept of *journey*. Let $\zeta(e,t)$ be the transfer delay of edge e at time t. A journey is similar to a path in $\mathcal{G}$, except that it accounts for link traversal times: a path $P_{\mathcal{G}}$ is a journey only if the message transfer delays between nodes ensures the traversal from u to v. More formally, a journey is a sequence of couples $\mathcal{J} = \{(e_1,t_1),...,(e_k,t_k)\}$ such that $\{e_1,...,e_k\}$ is a walk in G and $\forall 1 \leq i < k, e_i \in E_{t_i} \wedge t_{i+1} \geq t_i + \zeta(e_i,t_i)$. t_1 is denoted $departure(\mathcal{J})$ and $t_k + \zeta(e_k,t_k)$ is denoted $arrival(\mathcal{J})$. Intuitively, a *journey* is a path over time. As such, it has both a topological length (the length of the path in G) and a temporal length ($arrival(\mathcal{J}) - departure(\mathcal{J})$).

[BFJ03] also analyze the problem of least cost journeys, for different metrics:

- **Foremost journey:** the journey which arrives at the earliest possible time;
- **Shortest journey:** the journey with the the minimum topological length;
- **Fastest journey:** the journey with the minimum temporal length.

[CCF11] combines evolving graphs with the graph relabelings of [LMS01] in order to obtain a dynamic model which can abstract the communication models such as the message passing model or the shared memory model.

[FGA11] provides necessary and sufficient conditions for mutual exclusion and k-mutual exclusion in evolving graphs.

2.4.5 Time-Varying Graphs (TVG)

Time-Varying Graphs are a communication model introduced by Casteigts et al. in [CFQS12] with the goal of providing a very generic and modular model for dynamic networks, such that previous models can be simulated in it. Using definitions from the Evolving Graphs of [BF03], it defines a dynamic system as a TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ where:

- $V = \Pi$ is the set of nodes in the system;
- $E \subseteq V \times V$ is the set of edges (communication links);
- $\mathcal{T} = \mathbb{N}$ is a time span;
- $\rho : E \times \mathcal{T} \to \{0,1\}$ is the edge *presence* function, indicating whether an edge is active or not at a given time;
- $\zeta : E \times \mathcal{T} \to \mathbb{T}$ is the *latency* function, indicating the time taken by a message to cross a given edge if starting at a given time.

TVGs are used to model the communication graph of dynamic systems, including those presented in Sections 2.4.2 and 2.4.3. Additionally, the model offers optional parameters:

- $\psi : V \times \mathcal{T} \to \{0,1\}$ is the *node presence function*, indicating whether or not a node is present in the system at a given time;
- $\varphi : V \times \mathcal{T} \to \mathbb{T}$ is the *node latency function*, indicating the local processing time on a given node at a given time.

ψ allows the representation of processes joining and leaving (or failing) the system, and is necessary for the models defined in Sections 2.4.1 and 2.4.4.

The *underlying graph* $G = (V, E)$ is the footprint of $\mathcal{G}$, indicating which nodes have a relation at some time in $\mathcal{T}$. Most applications require G to be connected.

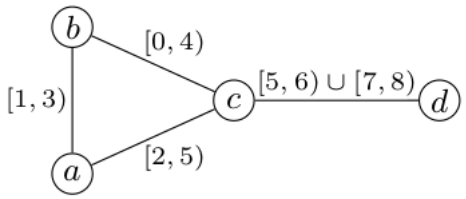


Figure 2.1: A simple TVG, with $V = \{a, b, c, d\}$. The intervals on each edge represent the periods of time when it is available and constitute the edge presence function ρ. This representation of the communication graph was initially introduced for evolving graphs in [BF03].

2.4.5.1 Journeys

The definition of a journey in [BF03] is adapted for the TVG model as follows: a journey is a sequence of couples $\mathcal{J} = \{(e_1, t_1), ..., (e_k, t_k)\}$ such that $\{e_1, ..., e_k\}$ is a walk in G and $\forall 1 \leq i < k, \rho(e_i, t_i) = 1 \wedge t_{i+1} \geq t_i + \zeta(e_i, t_i)$. If a *journey* exists between two nodes u and v, it is said that u can *reach* v, which is denoted $u \rightsquigarrow v$. The set of all journeys from node u to node v is denoted $\mathcal{J}^*_{u,v}$.

Additionally, a *direct journey* is a *journey* such that $\forall 1 \leq i < k, \rho(e_{i+1}, t_i + \zeta(e_i, t_i)) = 1$, meaning that it does not include any waiting time.

In [CFG+15], the expressivity of different variants of TVG is studied, in particular in regard to *direct* or *indirect journeys*.

2.4.5.2 TVG Classes

Various TVG connectivity assumptions are classified according to the level of graph connectivity they provide. A total of thirteen TVG classes are defined. Here are the definitions of some of the most important classes:

- **Class 1** $\exists u \in V : \forall v \in V, u \rightsquigarrow v$.
  There is a node that can reach all others.

- **Class 2** $\exists u \in V : \forall v \in V, v \rightsquigarrow u$.
  There is a node that can be reached by all others.

- **Class 3** (*Connectivity over time*): $\forall u, v \in V, u \rightsquigarrow v$.
  Every node can reach all others.

- **Class 4** (*Round connectivity*): $\forall u, v \in V, \exists \mathcal{J}_1 \in \mathcal{J}^*_{(u,v)}, \exists \mathcal{J}_2 \in \mathcal{J}^*_{(v,u)} : arrival(\mathcal{J}_1) \leq departure(\mathcal{J}_2)$.
  Every node can reach all others and be reached back afterwards.

- **Class 5** (*Recurrent connectivity*): $\forall u, v \in V, \forall t \in \mathcal{T}, \exists \mathcal{J} \in \mathcal{J}^*_{(u,v)} : departure(\mathcal{J}) > t$. Every node can reach all others infinitely often.

Figure 2.2 presents the relative strength of the different TVG classes. An arrow from one class to another indicates that the second class strictly includes the first one.

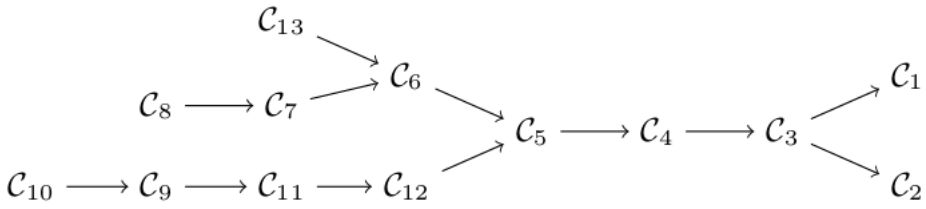


Figure 2.2: Relations of inclusion between TVG classes

The *T-interval connectivity* of [KLO10] (Section 2.4.2) is defined by class 10, whereas the *root components* of [BRS12] (Section 2.4.3) are stronger than class 1 but strictly weaker than class 3.

2.4.5.3 Timely Journeys

[GCLL15] adds to the TVG notions of Δ-*journey*, β-*journey* and (α, β)-*journey*. With $\Delta \in \mathbb{T}$, $\mathcal{J}$ is a Δ-*journey* at time t if and only if $\mathcal{J}$ is a *journey* and $t + arrival(\mathcal{J}) - departure(\mathcal{J}) \geq \Delta$. A β-edge is an edge such that latency is bounded by ζ_{MAX} and the edge is active for at least β time, a β-*journey* at time t is a Δ-*journey* such that:

- $\zeta_{\text{MAX}} < \beta \geq \Delta$.
- $\forall i \in [0, k), e_i$ is a β-edge.
- $\forall i \in [0, k), t_{i+1} \geq t_i + \beta$ (the times when edges are activated and their corresponding latencies allow a bounded sequential traversal).

A (α, β)-*journey* is a β-*journey* such that:

- The appearance of e_1 is bounded by $t_1 \leq t + \alpha$.
- $\forall i \in [1, k), t_{i+1} \leq t_i + \zeta(e_i, t_i) + \alpha$ (the appearances of the subsequent edges are also bounded by α).

These new definitions allow for the usage of partial synchrony assumptions in a TVG.

2.4.6 Unknown Asynchronous Dynamic Networks

This section regroups papers considering an asynchronous dynamic network, in the sense of an evolving communication graph, with unknown membership, meaning that even if n is finite, processes are not aware of its value and the identity of the participants.

[SAB+08] proposes a query-response-based implementation of failure detector $\diamond\mathcal{S}$ in a model where f is known from processes, along with the minimal cardinality d of the neighbourhood

of any node in the system. It does not use synchrony assumptions and requires behavioural properties instead:

- **Membership Property**: Let $range_i$ be the set of the neighbours of process p_i in the underlying static graph and K_i^t be the set of processes that received a query from p_i at time $t \in \mathcal{T}$. p_i satisfies the property if $\exists t \geq 0 \in \mathcal{T} : |K_i^t| > f + 1$.

- **Responsiveness Property**: Let $rec\_from_j^t$ the set of processes from which p_j has received responses from its query message that terminated at or before t. Process p_i verifies the property if $\exists u \in \mathcal{T} : \forall t > u, \forall p_j \in range_i, p_i \in rec\_from_j^t$.

[AST+10] introduces and implements the perfect partition participant detector $\diamond\mathcal{PD}$ which detects the nodes participating in stable partitions in a unknown directed dynamic network.

In [FT09], Friedman and Tcharny propose a failure detection protocol that probabilistically ensures completeness and accuracy properties in a mobile ad-hoc network. Each process can only communicate with the processes that are currently in its wireless range. The failure detection delay and the number of false suspicions depend on the network connectivity.

A model for unknown dynamic networks suited for implementations of failure detectors is presented in [GAS11, GSAS11]. It combines the *finite arrival* model of [Agu04] with the TVG of [CFQS12] (class 5, *recurrent connectivity*), and introduces the eventually strong failure detector with unknown membership $\diamond S^M$. If KNOWN is the set of processes known by any other process at any point in time, STABLE is the set of processes who never crash and FAULTY is the set of faulty processes, then $\diamond S^M$ is defined by the following properties:

- **Strong completeness**:

$$\exists t \in \mathcal{T}, \forall t' \geq t, \forall p_i \in \text{KNOWN} \cap \text{FAULTY} \Rightarrow \forall p_j \in \text{KNOWN} \cap \text{STABLE}, p_i \in \diamond S_{p_j}^M(t)$$

- **Eventual weak accuracy**:

$$\exists t \in \mathcal{T}, \forall t' \geq t, \exists p_i \in \text{KNOWN} \cap \text{STABLE} \Rightarrow \forall p_j \in \text{KNOWN} \cap \text{STABLE}, p_i \notin \diamond S_{p_j}^M(t)$$

The properties correspond to the original properties of $\diamond S$, but only consider processes in the KNOWN set. [GSAS12] presents an implementation of $\diamond S^M$ in the model of [GAS11] using a query-response mechanism along with message pattern assumptions. It relies on the following properties:

- **Stable Termination Property ($\mathcal{S}at\mathcal{P}$)**: Let p_i be a node which issues a query. Thus, $\exists p_j \in \text{STABLE}, p_j \neq p_i$, which receives that query.

- **Stable Responsiveness Property ($\mathcal{SRP}$)**: Let X_j^t be the set of processes from which p_j has received responses to its last query sent before t. $stable^t(p_i) \Leftrightarrow \forall t' > t, \psi(p_i, t') = 1$. $\mathcal{SRP}^t(p_i) \stackrel{\text{def}}{=} stable^t(p_i) \wedge \forall p_j \in \Pi, (\exists e_{i,j}, \exists t' \geq t, \rho(e_{i,j}, t') = 1) \Rightarrow \forall t'' \geq t' + \zeta(e_{i,j}, t'), p_i \in X_j^{t''}$.

$\mathcal{S}at\mathcal{P}$ guarantees that every query is received by at least one stable process. $\mathcal{SRP}$ ensures that eventually, all the neighbors of one stable process p_i will receive its response to every one of their query among the α_i first responses.

[AGSS13] presents an algorithm for Ω in the same model. It uses a similar query-response mechanism and depends on the $\mathcal{S}at\mathcal{P}$ and $\mathcal{SRP}$ properties, along with a TVG of class 5 (*recurrent connectivity*).

An algorithm for $\diamond S^M$ in the model of [GAS11] tolerating Byzantine failures is given in [GdLAS12].

In [TI15], Taheri and Izadi propose a protocol solving Byzantine consensus in an asynchronous dynamic system, using the necessary assumption that no more than $\lfloor \frac{n-1}{3} \rfloor$ processes are faulty.

Benchi et al. provided in [BLG15] an algorithm for consensus in asynchronous opportunistic networks with assumptions on the failure pattern.

[RAS15] introduces the initial knowledge-free quorum failure detector $\Sigma_\perp$ as a redefinition of Σ suited to unknown networks, where the intersection property cannot possibly be fulfilled from the beginning of the run. The quorums returned by $\Sigma_\perp$ can also contain the special value $\perp$, which is used as a marker to indicate that the minimum knowledge about process identities has not been reached yet and, therefore, the intersection property is not expected to hold yet. $\Sigma_\perp$ is defined by the following properties:

- **$\perp$-Limited intersection property**:

$$\forall t_1, t_2 \in \mathcal{T}, \forall p_1, p_2, \in \Pi, \Sigma_{\perp p_1}(t_1) \neq \perp, \Sigma_{\perp p_2}(t_2) \neq \perp : \Sigma_{\perp p_1}(t_1) \cap \Sigma_{\perp p_2}(t_2) \neq \emptyset$$

- **$\perp$-Extended completeness property**:

$$\exists \tau \in \mathcal{T}, \forall p \in \mathcal{C}, \forall t \in \mathcal{T}, t \geq \tau : \Sigma_{\perp p}(t) \neq \perp \wedge \Sigma_{\perp p}(t) \subseteq \mathcal{C}$$

Compared to the traditional properties of Σ, the intersection property only checks the intersection of non-$\perp$ quorums, and the completeness property ensures that eventually, $\perp$ is not returned anymore. The paper also gives an implementation of $\Sigma_\perp$ using message pattern assumptions similar to the $\mathcal{SRP}$ property, but regarding the distant communications inside a set of processes instead of the neighborhood of a single process.

2.4.7 Summary of Failure Detector Results in Unknown and/or Dynamic Systems

Table 2.3 summarizes the article presented in this chapter that studied failure detectors in unknown and/or dynamic systems. For each article, the table also specifies if the model used considers a dynamic system membership, a dynamic communication graph, and whether the system is unknown or not.

2.5 Conclusion

This chapter presented the failure detector results in the literature around the k-set agreement and FTME problems.

FD class	Article	Section	Dynamic membership	Dynamic graph	Unknown
Ω	[MRT+05]	2.4.1.1	Yes	No	Yes
Ω	[CRTW07]	2.4.1.1	Yes (Mobile Hosts), No (Support Stations)	Yes (MH) No (SS)	Yes (MH) No (SS)
Σ	[ADGF08]	2.4.1.1	Yes	No	Yes
$\Delta\Omega$	[LRAC12]	2.4.1.2	Yes	No	Yes
$\Delta\Omega$	[GLLR13]	2.4.1.2	Yes	Yes	Yes
$\diamond\mathcal{S}$	[SAB+08]	2.4.6	No	Yes	Yes
$\diamond\mathcal{PD}$	[AST+10]	2.4.6	Yes	Yes	Yes
$\diamond S^M$	[GSAS11]	2.4.6	Yes	Yes	Yes
Ω	[AGSS13]	2.4.6	Yes	Yes	Yes
$\Sigma_\perp$	[RAS15]	2.4.6	No	No	Yes

Table 2.3: A global view of failure detectors studied in unknown and/or dynamic systems

The $\Pi\Sigma_{x,y}$ failure detector is sufficient to solve k-set agreement problem in static and known message passing systems, provided that $k \leq xy$. Under this condition, it is the weakest failure detector to solve consensus and set agreement, and is weaker than other failure detectors for the cases of $2 \leq k < n-1$. Additionally, the $(\mathcal{T}, \Sigma^l)$ failure detector of [DGFGK05, BCJ09] is the weakest failure detector to solve FTME in static and known message passing systems.

Many dynamic system models have been proposed, but few failure detectors have been studied in such models, as seen in Table 2.3. In particular, the results around $\Pi\Sigma_{x,y}$ and $(\mathcal{T}, \Sigma^l)$ have not been extended to dynamic systems: this is, therefore, the focus of this thesis.

Chapter 3

A Failure Detector for k-Set Agreement in Unknown Dynamic Systems

Contents

3.1 System Model 30
3.1.1 Process Model 30
3.1.2 Communication Model 31
3.2 Failure Detectors for k-Set Agreement in Unknown Dynamic Systems 32
3.2.1 The $\Sigma_{\perp,k}$ Failure Detector 32
3.2.2 The Family of Failure Detectors $\Pi\Sigma_{\perp,x,y}$ 33
3.3 Assumptions 35
3.3.1 Time-Varying Graph Classes 35
3.3.2 Message Pattern Assumptions 37
3.3.3 Summary of Assumptions 40
3.3.4 Implementation of Message Pattern Assumptions 40
3.3.5 Comparable Assumptions in the Literature 41
3.4 Failure Detector Algorithms 42
3.4.1 An Algorithm for $\Sigma_{\perp,k}$ 42
3.4.2 An Algorithm for $\Pi\Sigma_{\perp,x}$ 46
3.4.3 An Algorithm for $\Pi\Sigma_{\perp,x,y}$ 50
3.5 A k-Set Agreement Algorithm 50
3.5.1 The $Alpha_x$ Sub Protocol 50
3.5.2 $Alpha_x$ Algorithm 51
3.5.3 k-Set Agreement Algorithm 53
3.6 Conclusion 54

This chapter focuses on the k-set agreement problem, which is a generalization of the consensus problem such that 1-set agreement is consensus. In the k-set agreement problem, each process proposes a value, and some processes eventually decide a value while respecting the properties

of validity (a decided value is a proposed value), termination (every correct process eventually decides a value) and agreement (at most k values are decided).

As described in Section 2.4, protocols solving consensus or k-set agreement have been proposed for dynamic systems, but they assume synchronous communications (as in [BRS12, BRS$^+$18, KMO11, SS14]) or make strong assumptions on the number of process failures [BLG15].

The $\Pi\Sigma_{x,y}$ failure detector (presented in Section 2.3.2.8) was introduced in [MRS12] and is sufficient to solve k-set agreement in static networks (if and only if $k \geq xy$) while being weaker than other known failure detectors which solve the same problem. However, this failure detector relies on information about the set of participants which is not available in unknown networks. Additionally, traditional failure detectors rely on a full connectivity of the network graph, which is not available in a dynamic network.

In this chapter, the definition of $\Pi\Sigma_{x,y}$ is extended in order to obtain the $\Pi\Sigma_{\perp,x,y}$ failure detector, which is capable of solving k-set agreement in unknown dynamic systems, and provide implementations of this new detector. Additionally, this chapter provides an adaptation of the k-set agreement algorithm of [BT10, MRS12] to solve k-set agreement using $\Pi\Sigma_{\perp,x,y}$.

The model assumptions proposed to implement $\Pi\Sigma_{\perp,x,y}$ are generic and expressed in terms of message pattern, which allows the model to be applied to a range of systems. Concrete examples of partial synchrony and failure pattern properties which are sufficient to ensure the more generic message pattern assumptions are also presented.

The system is modeled using the formalism of the Time-Varying Graph (TVG), as defined in [CFQS12] (see Section 2.4.5).

This chapter thus brings the following main contributions:

1. The definition of the $\Pi\Sigma_{\perp,x,y}$ failure detector as an adaptation of $\Pi\Sigma_{x,y}$ to solve k-set agreement in unknown dynamic systems (Section 3.2).

2. An algorithm implementing $\Pi\Sigma_{\perp,x,y}$ in unknown dynamic systems, with connectivity and message pattern assumptions (Section 3.4).

3. An algorithm solving k-set agreement in unknown dynamic systems enriched with $\Pi\Sigma_{\perp,x,y}$ (Section 3.5).

3.1 System Model

3.1.1 Process Model

A finite set of n processes $\Pi = \{p_1, ...p_n\}$ participate in the system. *The processes are synchronous* (there is a bound on the relative speed of processes) and uniquely identified, although initially they are only aware of their own identities. Processes are not required to know the value of n.

A run is a sequence of steps executed by the processes while respecting the causality of operations (each received message has been previously sent). Processes can join and leave the system during the run (Π is the set of all processes that participate in the system at some point in time). Processes may also crash, and no difference is made between a process that crashes permanently and a process that leaves the system permanently: in both cases the process is considered *faulty* in that run. A process that is not faulty is called *correct*. Note that this

definition of faulty and correct processes is not exactly the traditional one. Indeed, correct processes can crash or leave the system, as long as they recover or come back later. Only processes that crash or leave permanently are considered faulty.

Correct processes can leave the system and come back infinitely often, but they can only crash and recover a finite number of times. The critical difference is that a process that leaves the system keeps its memory intact, whereas a crashed process does not.

The set of all correct processes is called $\mathcal{C}$. There is a bound $f < n$ on the number of faulty processes in a run.

3.1.2 Communication Model

Processes communicate by sending and receiving messages. *Communications are asynchronous*: there is no bound on message transfer delays. Therefore, even though processes are synchronous, they do not cooperate in a synchronous way.

3.1.2.1 Time-Varying Graphs

The system is dynamic, which means that nodes and communication links can appear or disappear during the run: therefore, the communication graph will change over time. The usual notion of path in the graph is not sufficient to define reachability in such a dynamic graph. To solve this issue, in this chapter, the Time-Varying Graph (TVG) formalism defined by Casteigts et al. in [CFQS12] (Section 2.4.5) is used to model the communication graph.

Note that processes do not know the values of the ζ latency function, which is only introduced for the simplicity of presentation. Since communications are asynchronous, the values of ζ are finite but not necessarily bounded.

Consider the following example: a graph where $E = V \times V$ and every edge in the system is active infinitely often (longer than the message transfer time), but no more than one edge is ever active at a time. In such a system, there are journeys infinitely often between every node and the connectivity is sufficient to solve complex problems such as consensus. However, at any given instant, the graph is partitioned into at least $n - 1$ independent subsets. This shows that similarly to paths, the usual notion of graph partitioning loses relevancy in TVGs, since the number of partitions at a particular instant in the run is not a very useful parameter. Instead, the number of *partitions over time* is relevant. In the rest of the chapter, the word *partition* is used to refer to a subset of the network that is isolated from the rest of the network for an arbitrarily long duration, and not just temporarily.

3.1.2.2 Communication Primitive

Processes communicate exclusively by sending messages with a very simple broadcast primitive. When a process p_i calls the broadcast primitive, the message is simply sent to the processes that are currently in p_i's neighborhood, including p_i. The broadcast is not required to provide advanced features such as message forwarding, routing, message ordering or any guarantee of delivery.

3.1.2.3 Channels

Channels are fair-lossy. Messages may be lost but, if the edge is active for the entire time of the message transfer, a message sent infinitely often will be received infinitely often. Messages may be duplicated, but a message may only be duplicated a finite number of times. No message can be created or altered. No assumption is made on message ordering and channels are not required to be FIFO.

3.2 Failure Detectors for k-Set Agreement in Unknown Dynamic Systems

The goal of this chapter is to adapt existing failure detectors to solve k-set agreement in dynamic systems.

In order to deal with the dynamics of the system, some connectivity properties are required in addition to the information on failures provided by the failure detector.

Furthermore, the system model considers an unknown network where processes have no information on system membership at the beginning of the run. A way to circumvent this issue was proposed in [RAS15] in the form of the $\Sigma_\perp$ failure detector (see Section 2.4.6).

The $\Pi\Sigma_{x,y}$ failure detector of [MRS12] (see Section 2.3.2.8) will be augmented with connectivity properties and extended with the method of [RAS15] in order to obtain a failure detector sufficient to solve the k-set agreement problem in unknown dynamic systems.

3.2.1 The $\Sigma_{\perp,k}$ Failure Detector

As explained in Sections 2.3.2.4 and 2.3.2.5, the intersection property of both Σ and Σ_k must hold over time, which means that if a process queries its failure detector before any communication has taken place, the returned quorum must intersect with the quorums formed by processes later in the run. In known networks, implementations of Σ traditionally solve this issue by returning Π as a quorum at the beginning of the run [DFG10]. This is not an option in unknown networks where system membership knowledge is only established through communication.

The $\Sigma_\perp$ failure detector ([RAS15]) is an adaptation of Σ for unknown networks. Instead of returning a quorum, $\Sigma_\perp$ can also output the default value $\perp$ whenever the knowledge necessary to form a quorum has not been gathered yet.

In order to solve k-set agreement in unknown dynamic networks, $\Sigma_{\perp,k}$ failure detector is defined to combine the properties of Σ_k and $\Sigma_\perp$. It also includes a connectivity property which replaces (and is weaker than) the assumption of a static and complete network.

The $\Sigma_{\perp,k}$ failure detector provides each process p_i with a quorum denoted qr_i^τ (which is either a set of process identities or the special value $\perp$) at any time instant τ.

The following definition is introduced for the convenience of the presentation: definition:

Definition 3.1 (Recurrent neighborhood)**.** *The recurrent neighborhood of a correct process p_i, denoted R_i, is the set of all correct processes whose quorums intersect infinitely often with p_i's quorums.* $\forall p_i \in \mathcal{C}, R_i = \{p_j \in \mathcal{C} \mid \forall \tau, \exists \tau_i, \tau_j \geq \tau : qr_i^{\tau_i} \neq \perp \wedge qr_j^{\tau_j} \neq \perp \wedge qr_i^{\tau_i} \cap qr_j^{\tau_j} \neq \emptyset\}$.

Note that $p_j \in R_i$ is an equivalence relation between p_i and p_j. By definition, $\forall p_i \in \mathcal{C} : p_i \in R_i$, therefore $R_i \neq \emptyset$.

A correct process p_i can *reach* another correct process p_j if, provided that p_i sends messages infinitely often, p_j receives them infinitely often.

$\Sigma_{\perp,k}$ is defined by the self-inclusion, quorum liveness, quorum intersection and quorum connectivity properties.

- **Self-inclusion**: Every process includes itself in its non-$\perp$ quorums.

$$\forall p_i \in \Pi, \forall \tau : (qr_i^\tau \neq \perp) \implies (p_i \in qr_i^\tau) \ .$$

- **Quorum liveness**: Eventually, every correct process stops returning $\perp$ and its quorums only contain correct processes.

$$\exists \tau, \forall p_i \in \mathcal{C}, \forall \tau' \geq \tau : qr_i^{\tau'} \neq \perp \wedge qr_i^{\tau'} \subseteq \mathcal{C} \ .$$

- **Quorum intersection**: Out of any $k+1$ non-$\perp$ quorums, at least two intersect.

$$\begin{gathered}\forall \tau_1, ..., \tau_{k+1} \in \mathcal{T}, \forall id_1, ..., id_{k+1} \in \Pi, \\ \exists i, j : 1 \leq i \neq j \leq k+1 : \\ (qr_{id_i}^{\tau_i} \neq \perp \wedge qr_{id_j}^{\tau_j} \neq \perp) \implies (qr_{id_i}^{\tau_i} \cap qr_{id_j}^{\tau_j} \neq \emptyset) \ .\end{gathered}$$

- **Quorum connectivity**: Every correct process p_i can *reach* every process in R_i.

As presented in Sections 2.3.2.5 and 2.4.6, Σ_k and $\Sigma_\perp$ were defined with only 2 properties (liveness and intersection). Self-inclusion is a property added to Σ_x and $\Pi\Sigma_x$ by the authors in [MRS12] for the sake of the simplicity of algorithm proofs, and it is trivially implemented by the algorithms presented in this chapter. Quorum connectivity is the property added to deal with network dynamics.

Intuitively, the quorum connectivity property means that processes belong to the same partition as their recurrent neighborhood. Note that $\forall p_i, p_j \in \mathcal{C} : p_i \in R_j \implies p_j \in R_i$, thus quorum connectivity enables two-way communication between p_i and p_j. This property is not very costly, since most failure detector implementations already require some level of connectivity between processes in a quorum in order to form the quorums themselves. This is the case for the $\Sigma_{\perp,k}$ algorithm presented in Section 3.4, which does not require any additional assumption to implement quorum connectivity.

3.2.2 The Family of Failure Detectors $\Pi\Sigma_{\perp,x,y}$

$\Pi\Sigma_{\perp,x,y}$ is defined as an extension of $\Pi\Sigma_{x,y}$ that includes the properties of $\Sigma_{\perp,x}$ and is capable of solving k-set agreement in unknown dynamic systems.

Similarly to [MRS12], $\Pi\Sigma_{\perp,x,y}$ is defined incrementally: $\Pi\Sigma_{\perp,x}$ is defined firstly.

3.2.2.1 The Failure Detector $\Pi\Sigma_{\perp,x}$

At any time instant τ, $\Pi\Sigma_{\perp,x}$ provides each process p_i with a quorum denoted qr_i^τ (which is either a set of process identities or the special value $\perp$) and a leader denoted $leader_i^\tau$ (which is a process identity).

$\Pi\Sigma_{\perp,x}$ is defined by the following properties:

- Self-inclusion
- Quorum liveness
- Quorum intersection
- Quorum connectivity

$\}\ \Sigma_{\perp,x}$

- Eventual partial leadership

First, an eventual partial leader is defined as follows:

Definition 3.2 (Eventual partial leader)**.** *An eventual partial leader p_l is a correct process such that every process in the recurrent neighborhood of p_l eventually recognizes p_l as its leader forever.* $p_l \in \mathcal{C} \wedge \forall p_i \in R_l, \exists \tau, \forall \tau' \geq \tau : leader_i^{\tau'} = p_l$.

The set of all eventual partial leaders is denoted L.

- **Eventual partial leadership**: For every correct process p_i, there is an eventual partial leader p_l that can *reach* p_i.

The original eventual partial leadership property used in [MRS12] simply requires the existence of an eventual partial leader in the system. The new version of the property similarly implies that $L \neq \emptyset$ (since $\mathcal{C} \neq \emptyset$), but also implies that each correct process must be reachable by one eventual partial leader (which, depending on the level of connectivity, may require more than one leader). In a static and connected network, both properties are equivalent: a single eventual partial leader is necessary and sufficient to fulfill the property, since the connected communication graph enables this single leader to reach every correct process.

In a k-set agreement algorithm, the eventual partial leaders are those processes that eventually decide. In order to ensure termination, the deciding leaders must, therefore, be able to inform the rest of the system of their decision. However, in a dynamic network, the mere existence of an eventual partial leader does not provide the latter with the necessary connectivity to guarantee termination. This is why in dynamic networks, the new eventual partial leadership property is stronger than the original one and imposes the required connectivity.

The eventual partial leadership property implies a trade-off between the number of eventual partial leaders in the system and graph connectivity. On the one hand, if there is a single leader in the system, then this leader must be able to reach every correct process in the system. On the other hand, if the communication graph is partitioned, then there must be at least one local leader per partition.

Such a trade-off implies that the eventual partial leadership property does not prevent the system from being partitioned into up to n partitions over time, provided that every correct

process identifies itself as its own eventual partial leader. However in this scenario it would be impossible to verify the quorum intersection and quorum connectivity properties.

3.2.2.2 The Failure Detector $\Pi\Sigma_{\perp,x,y}$

The definition of $\Pi\Sigma_{\perp,x,y}$ is the same as $\Pi\Sigma_{x,y}$ in [MRS12], except that it uses $\Pi\Sigma_{\perp,x}$ instead of $\Pi\Sigma_x$. $\Pi\Sigma_{\perp,x,y}$ can be seen as y instances of $\Pi\Sigma_{\perp,x}$ running concurrently.

$\Pi\Sigma_{\perp,x,y}$ provides each process p_i with an array $FD_i[1..y]$ such that for each j, $1 \leq j \leq y$, $FD_i[j]$ is a pair containing a quorum $FD_i[j].qr$ and a process index $FD_i[j].leader$. The array satisfies the following properties:

- **Vector safety**: $\forall j \in [1..y] : FD_i[j].qr$ satisfies the self-inclusion, liveness, intersection and quorum connectivity properties of $\Pi\Sigma_{\perp,x}$.
- **Vector liveness**: $\exists j \in [1..y] : FD_i[j]$ satisfies the eventual partial leadership property of $\Pi\Sigma_{\perp,x}$.

The idea is to reduce the cost of the system assumptions: the liveness property only needs to be verified by one out of a set of y instances of the detector.

The authors in [MRS12] prove that for $1 \leq y \leq n-1$, $\Pi\Sigma_{1,y}$ is as strong as $\langle \Sigma_1, \overline{\Omega}_y \rangle$. This shows that the y parameter of $\Pi\Sigma_{x,y}$ (and $\Pi\Sigma_{\perp,x,y}$) is comparable to the y of $\overline{\Omega}_y$.

$\Pi\Sigma_{\perp,x,y}$ is sufficient to solve the k-set agreement problem in unknown dynamic systems if $k \geq xy$. This will be proved by providing a k-set agreement algorithm relying on $\Pi\Sigma_{\perp,x,y}$ in Section 3.5.

3.3 Assumptions

This section presents three system assumptions. The algorithms presented in Section 3.4 will then list the assumptions from this section on which they rely.

3.3.1 Time-Varying Graph Classes

In addition to defining the formalism of the TVG, Casteigts et al. present in [CFQS12] a number of TVG classes which provide different levels of connectivity assumptions. Class 5 is particularly relevant to this chapter:

Definition 3.3 (Class 5: recurrent connectivity [CFQS12])**.** *All processes can reach each other infinitely often through journeys.* $\forall u, v \in \Pi, \forall \tau, \exists \mathcal{J} \in \mathcal{J}^*_{(u,v)} : departure(\mathcal{J}) > \tau$.

This connectivity assumption does not exactly fit the requirements of the proposed algorithms. On the one hand, it is too strong. It implies a global connectivity between any two processes in the system, which is not necessary to solve k-set agreement, since the problem can be solved in a system partitioned into k subsets. On the other hand, class 5 is too weak since it relies on the notion of journey, which is insufficient to ensure the transmission of messages. Even if a journey exists between p_i and p_j, there is no guarantee that a message sent by p_i can reach p_j. In fact, even if the edge between p_i and p_j is active infinitely often and the message is

sent infinitely often, the message might always be sent in between two activation periods of the edge, thus never crossing it. To solve this problem, Gómez-Calzado et al. defined in [GCLL15] the notion of timely journeys for the case of synchronous systems. This solution can be extended into γ-journeys for the case of asynchronous communications.

Definition 3.4 (γ-Journey)**.** *A γ-journey $\mathcal{J}$ (where $\gamma > 0$ is a time duration) is a journey such that every node on the path can wait up to γ units of time after the next edge becomes active before forwarding the message. Since the message may be sent at any time within the γ time window and the channel latency may vary during that time, the edge must remain active long enough for the worst case duration.*
- $\forall i, 1 \leq i \leq |\mathcal{J}|$, *$e_i$ stays active from time t_i until, at least, time* $t_i + max_{0 \leq j \leq \gamma}\{j + \zeta(e_i, t_i + j)\}$.
- $\forall i, 1 \leq i < |\mathcal{J}|, t_{i+1} \geq t_i + max_{0 \leq j \leq \gamma}\{j + \zeta(e_i, t_i + j)\}$.

With a γ-journey, processes are given an additional time window of γ units of time to send the message. In [GCLL15], this time was used to detect the activation of the edge. This solution is appropriate for point-to-point communications in a known network, since it allows the sender of the message to resend the message to the receiver whenever the edge appears again. However, this is not helpful in an unknown non-complete network where processes have to rely on blind broadcasts and forwarding to propagate information.

Instead, the time window provided by γ-journeys is used as an upper bound on the time between two transmissions of the message. This explains the need for synchronous processes: each process should be able to repeatedly send every message at least once every γ units of time.

Provided that processes receive their own broadcasts within γ units of time and then re-broadcast it, it is ensured that every message is sent at least once every γ units of time. If there is infinitely often a γ-journey from p_i to p_j, then p_i can reach p_j.

The set of all the γ-journeys from u to v is called $\mathcal{J}^{\gamma}_{(u,v)}$.

Using class 5 as a starting point, TVGs of class 5-(α, γ) are defined as follows. γ is the time duration parameter of γ-journeys, and α is a parameter defining the number of correct processes that each correct process is ensured to communicate with.

Assumption 3.1 (Class 5-(α, γ): (α, γ)-recurrent connectivity)**.** *Every correct process can reach and be reached through γ-journeys infinitely often by at least α correct processes.*

$$\forall p_i \in \mathcal{C}, \exists P_i \subseteq \mathcal{C}, |P_i| \geq \alpha, \forall t \in \mathcal{T}, \forall p_j \in P_i,$$
$$\exists \mathcal{J}_i \in \mathcal{J}^{\gamma}_{(p_i,p_j)} : departure(\mathcal{J}_i) \geq t \ \wedge$$
$$\exists \mathcal{J}_j \in \mathcal{J}^{\gamma}_{(p_j,p_i)} : departure(\mathcal{J}_j) \geq t \ .$$

This assumption is parametrized by the two values α and γ. A low γ value weakens the connectivity assumption by allowing shorter time windows for the journeys, but implies that processes must be able to send messages more often to ensure that a message is sent within the shorter window. On the other hand, a high γ value reduces the number of journeys that are qualified as γ-journeys, thus strengthening the connectivity assumption, but accepts slower processes.

The α parameter also presents a trade-off: class 5-(α, γ) indirectly implies that there must be at least α correct processes in the system. As a result, a high α value will result in a strong

assumption on the number of process failures which can be costly in a dynamic system. A low α value would strengthen the message pattern assumptions presented in the next section.

Class 5-(α, γ) also implies that all correct processes must know a lower bound for α.

To summarize, the assumption of a TVG of class 5-(α, γ) means that correct processes are able to communicate infinitely often with a subset of α correct processes. This property ensures that correct processes will not wait for messages forever, which enables the failure detector algorithm to ensure the quorum liveness property. Additionally, if the algorithm ensures that every correct process p_i eventually only forms quorum from the P_i set, then class 5-(α, γ) also ensures quorum connectivity.

3.3.2 Message Pattern Assumptions

This section presents message pattern assumptions, as defined by Mostéfaoui et al. in [MMR03]. The message pattern model consists in assuming some properties on the relative order of message deliveries. If processes periodically wait for a certain number of messages, the idea is to assume that the message sent by some specific process will periodically be among the first ones to be received.

In order to express the message pattern assumptions used to implement $\Pi\Sigma_{\perp,x,y}$, it is necessary to assume that the distributed algorithm executed by processes uses a query-response mechanism. Processes periodically issue query messages, to which other processes respond.

The principle of the failure detector algorithm revolves around processes repeatedly issuing a query and then waiting for responses from α processes. The α parameter is therefore the minimum size of quorums returned by the algorithm, which does not necessarily constitute an assumption on the number of failures, since α might be equal to 1. Note that α is the same parameter used to define TVGs of class 5-(α, γ) which ensures that correct processes will not wait for messages infinitely.

The first α processes whose response to a given query from process p_i are received by p_i are called a *response set*.

3.3.2.1 Assumption for Quorum Intersection

The assumption of a TVG of class 5-(α, γ) is not sufficient to ensure the quorum intersection property. In [BT10], Bouzid and Travers proposed a method to implement quorums: if processes repeatedly wait for messages from at least $\lfloor \frac{n}{k+1} \rfloor + 1$ processes before outputting these processes as their new quorum, then the size of quorums alone is sufficient to ensure intersection. This method implies that there must be at least $\lfloor \frac{n}{k+1} \rfloor + 1$ correct processes in the system, otherwise processes would wait forever, thus preventing liveness.

In a dynamic system where processes are expected to join and leave the system, an assumption on the number of process failures seems too costly. For this reason, the failure detector algorithms presented in the next section rely on the message pattern approach.

The following assumption is sufficient for the $\Pi\Sigma_{\perp,x,y}$ algorithm to implement quorum intersection. It was obtained by generalizing the assumption used for the case $k = 1$ in [RAS15].

Assumption 3.2 (Generalized winning quorums). *$\exists m \in [1, k]$ and $\exists Q_{w1}, ..., Q_{wm} \subseteq \Pi$ (called winning quorums). Each winning quorum Q_{wi} is associated with a number $w_i \geq 1$ (called the* weight *of Q_{wi}) such that $\sum_{i=1}^{m} w_i \leq k$. $\forall p \in \Pi$, every time p issues a new query, $\exists i \in [1, m]$ such that $Q_{wi} \neq \emptyset$ and out of the first α processes from which p receives a response, at least $\lfloor \frac{|Q_{wi}|}{w_i+1} \rfloor + 1$ of them are in Q_{wi}.*

Intuitively, Assumption 3.2 requires that there are m sets of processes, the winning quorums, that answer faster than others, i.e., faster enough for subsets of these sets to be always included in every response set. In addition, every time a correct process issues a query, connectivity must allow for a subset of one of these winning quorums to receive and respond to the query. Note that winning quorums do not necessarily correspond to quorums returned by the failure detector at some point: instead they are sets of processes that have a tendency to be included in response sets.

The *weight* w_i of a winning quorum is a parameter which states which proportion of the winning quorum must be included in response sets. A winning quorum of weight 1 must be included in strict majority in a response set, whereas winning quorums of higher weights can be included in smaller proportions. The sum of all winning quorum weights is limited by k.

It is interesting to consider some extreme instances of this assumption. The first extreme is $m = k$. In this particular case, all winning quorums are necessarily of weight 1, and therefore each response set must include a strict majority of one of the winning quorums. Since each response set includes the strict majority of one out of k winning quorums, it is easy to see that out of any $k + 1$ response sets, at least two will necessarily intersect.

Fig. 3.1 shows an example for $m = k = 3$ in which winning quorums are represented by dashed red circles. Each solid black circle represents a response set. Note that out of any 4 response sets, at least 2 intersect.

Another extreme case is $m = 1$ and $w_1 = k$. In this particular case, all response sets must contain a small part of a single winning quorum.

Fig. 3.2 shows an example for $m = 1$ and $k = 3$. Similarly to Fig. 3.1, the winning quorum is represented by a dashed red circle, and response sets are represented by solid black circles. Once again, 2 out of any 4 response sets intersect.

The flexibility in the second example lies in which subset of the winning quorum will be included in each response set, while the flexibility in the first example lies in which winning quorum would be included in majority by each response set.

Assumption 3.2 implies that there is at least one winning quorum Q_{wi} such that at least $\lfloor \frac{|Q_{wi}|}{w_i+1} \rfloor + 1$ of the processes in Q_{wi} are correct. If $\alpha = \lfloor \frac{|Q_{wi}|}{w_i+1} \rfloor + 1 = 1$, there is no assumption on the number of failures but $|Q_{wi}| < w_i + 1$, which leaves minimal flexibility on the processes that must be included in every response set, thus strengthening the message pattern assumption. On the other hand, if $|Q_{wi}|$ (and therefore α) is high, the number of failures is limited but each response set must contain a subset of a larger set, which allows for more flexibility in the message pattern.

3.3.2.2 Assumption for Eventual Partial Leadership

In order to ensure the eventual partial leadership property, processes need to identify a local leader. Since the eventual partial leadership property is supposed to be implemented on top of $\Sigma_{\perp,x}$, the notion of quorum can be used to define this new assumption.

Additionally, the order of processes in quorums is used to single out the leader. For this purpose, processes in a quorum are assumed to be totally ordered. Any specific ordering can be used. A natural choice would be to use the order in which the processes were added to the quorum. Another simple choice would be to order according to process identifiers. For a process p_i and a quorum qr, if $p_i \in qr$, then $pos(p_i, qr)$ denotes the position of p_i in qr according to the chosen total order. If p_i is the first process in qr_j^τ, then $pos(p_i, qr_j^\tau) = 1$. In this particular case, p_i is said to be the *candidate* of p_j at time τ.

Eventual partial leaders are defined as follows:

Definition 3.5 (Eventually winning process)**.** *A correct process p_l is called an eventually winning process if there is a time τ such that after $\tau, \forall \tau' \geq \tau, \forall p_i \in R_l \backslash \{p_l\}$:*
1) p_l is present in every quorum formed by p_i. $p_l \in qr_i^{\tau'}$.
2) p_l's identity is always positioned in p_i's quorum before the identities of other processes in R_i. $\forall p_j \in R_i \backslash \{p_l, p_i\} : pos(p_l, qr_i^{\tau'}) < pos(p_j, qr_i^{\tau'})$.
3) In every quorum formed by p_i, there is another process that also belongs to R_l. $\exists p_j \in R_l \backslash \{p_l, p_i\}$: $p_j \in qr_i^{\tau'}$.

Point (1) means that after some time, p_l must be fast enough to ensure that its responses arrive in time to take part in every local quorum.

The implication behind (2) depends on the chosen ordering method. If processes are ordered by date of addition to the quorum, then (2) implies that after some time, p_l must be faster than the rest of the recurrent neighborhood of p_i. If processes are ordered by process identities, p_l must have the smallest process identity in the recurrent neighborhood.

It is easy to see how (1) and (2) can be used: if p_l belongs to every quorum and is singled out by the quorum order, processes in R_l can reliably select their candidate as leader.

Note that (2) excludes the case $p_i = p_j$, since otherwise p_l would have to be placed before p_i in p_i's quorums. If processes are ordered by date of addition to the quorum, this expectation would be very unrealistic since receiving its own message is a local computation and should therefore be faster than receiving p_l's message.

Point (3) requires that processes in R_l must not only communicate with p_l but also with each other to some extent, which enables them to share the information that p_l is their candidate. Note that (3) also requires the processes p_l, p_i and p_j to be distinctly defined: therefore, in order for an eventually winning process to exist, there must be at least 3 correct processes in the system ($f \leq n - 3$). Since p_l, p_i and p_j must be included in the same quorums, α must also be equal to 3 or greater.

The set of all eventually winning processes is called W.

The following assumption enables the failure detector algorithm to ensure the eventual partial leadership property.

Assumption 3.3 (Eventually winning γ-sources)**.** *For every correct process p_i, there is an eventually winning process p_l such that there is infinitely often a γ-journey from p_l to p_i.* $\forall p_i \in \mathcal{C}, \exists p_l \in W, \forall \tau : \exists \mathcal{J} \in \mathcal{J}^{\gamma}_{(p_l,p_i)} \wedge departure(\mathcal{J}) > \tau$.

The $\Pi\Sigma_{\perp,x}$ algorithm will ensure that eventually winning processes are eventual partial leaders. As a result, this assumption will be sufficient to ensure the eventual partial leadership property.

3.3.3 Summary of Assumptions

Table 3.1 summarizes the assumptions presented in this section and the failure detector properties that rely on them for implementation.

Table 3.1: Assumptions for failure detector implementations

Assumption	Failure detector property
Assumption 3.1	$\Sigma_{\perp,k}$: quorum liveness $\Sigma_{\perp,k}$: quorum connectivity
Assumption 3.2	$\Sigma_{\perp,k}$: quorum intersection
Assumption 3.3	$\Pi\Sigma_{\perp,x}$: eventual partial leadership

The self-inclusion property of $\Sigma_{\perp,k}$ is absent from this table because it does not require any assumption and will simply be ensured through algorithmic properties.

3.3.4 Implementation of Message Pattern Assumptions

Assumptions 3.2 and 3.3 are very abstract and it can be difficult to judge at first glance how likely they are of being verified in a real network. This is because the assumptions presented in this section are meant to be as close as possible to the minimum model strength required to ensure that the algorithm implements the $\Pi\Sigma_{\perp,x,y}$ failure detector. The message pattern model enables such an implementation while keeping the model generic and applicable to different networks. In the following, examples of more traditional assumptions that are sufficient to ensure Assumptions 3.2 and 3.3 are presented.

3.3.4.1 Implementation of Assumption 3.2

A simple and intuitive method is to assume that $|\mathcal{C}| \geq \lfloor \frac{n}{k+1} \rfloor + 1$. In this case, Assumption 3.2 is trivially verified with $m = 1, w_1 = k$ and $Q_{w1} = \Pi$. This implies that $\alpha \geq \lfloor \frac{n}{k+1} \rfloor + 1$, and, thus, the minimal size of quorums is sufficient to ensure intersection. This particular case is the method used to implement Σ_k in static networks in [BT10].

Another approach would be to use a partial synchrony assumption. For a given duration Δ, let us call Δ-journey a γ-journey $\mathcal{J}$ such that $arrival(\mathcal{J}) - departure(\mathcal{J}) \leq \Delta$. Π is then separated into two subsets: slow processes and fast processes. A slow process p_i is a process such that there is never a Δ-journey from p_i to any correct process $p_j \in \mathcal{C} \backslash \{p_i\}$. Fast processes are all other processes and Q is the set of all correct fast processes. The assumption is that for any

correct process p_i and at any time, there are Δ-journeys linking p_i to at least $\lfloor \frac{|Q|}{k+1} \rfloor + 1$ processes from Q. Assumption 3.2 is verified with $m = 1, w_1 = k$ and $Q_{w1} = Q$.

3.3.4.2 Implementation of Assumption 3.3

One way to ensure Assumption 3.3 is that there is a correct subset Q of the system that is constantly connected and recognizes a leader $p_l \in Q$, that can reach the entire system infinitely often. The leader must be known from the other processes in Q from the start (it can simply be the lowest process identifier in Q, for example). When a process in Q issues a query, the communication layer for that process will then wait for a response from p_l and a response from another process in Q before delivering any other response. This is sufficient to ensure that p_l is the first process in every quorum formed in Q, and that processes in Q communicate with each other to a sufficient extent.

3.3.4.3 Practical Issues

From a practical point of view, some types of networks are particularly adapted to ensure Assumptions 3.2 and 3.3. In wireless mesh networks ([AWW05]), the nodes move around a fixed set of nodes and each mobile node eventually connects to a fixed node. Wireless sensor networks ([ASSC02]) can be organized in clusters; one node in each cluster is designated the cluster head. Messages sent between clusters are routed through the cluster heads of the sending and receiving clusters. An infra-structured mobile network ([CRTW07]) is composed of Mobile Hosts (MH) and Mobile Support Stations (MSS). A MH is connected to a MSS if it is located in its transmission range, and two MHs can communicate only through MSSs.

In each of these network models, there is a privileged subset of powerful nodes (fixed nodes, cluster heads, MSSs) that can be used as a winning quorum to satisfy Assumption 3.2 or as the neighborhood R_l of an eventual winning process p_l for Assumption 3.3.

Both assumptions can also be ensured from a probabilistic perspective. If a subset Q of the system is made of powerful nodes that respond to queries much faster than the rest of the nodes, then there is a high probability that Assumption 3.2 will be verified. Similarly, Assumption 3.3 can be verified in a probabilistic way with a leader that is simply a powerful process benefiting from very small communication delays with the processes around it.

3.3.5 Comparable Assumptions in the Literature

This section attempts to put the strength of Assumptions 3.2 and 3.3 into perspective by comparing them to some other existing models.

In [AG13], Afek and Gafni propose an implementation of read and write operations in a dynamic synchronous message passing system. Although the underlying network is assumed to be complete, in each synchronous round a subset of edges lose their messages. Therefore, such a system can be modeled as a TVG where the edges that successfully deliver their message in a round are considered active in that round. As a result, the message adversary that decides which messages will go through can be compared to a connectivity assumption. The paper defines the Traversal Path (TP) adversary as a model in which, for every synchronous round, the directed graph defined by the successfully delivered messages in this round contains a directed

path passing through all the nodes. This connectivity assumption is weaker than a TVG of class 5, because traversal paths are directed paths, which implies that every process can not necessarily communicate with every other. The comparison with class 5-(α, γ) is less straightforward. On the one hand, class 5-(α, γ) implies two-way connectivity whereas a TP adversary only requires one-way connectivity. On the other hand, class $5(\alpha, \gamma)$ only requires connectivity between a limited number of nodes (as defined by the α parameter) and allows network partitioning, whereas a TP adversary connects the entire system.

In [BRS14], Biely et al. define and implement the generalized loneliness failure detector $\mathcal{L}_k$ in a static and connected network. For this purpose, the authors use the $M^{anti(x)}$ message pattern model, which is defined by the x-Anti-Source. An x-Anti-Source is a process which is ensured to receive responses from x processes to every query it issues before it issues the next query. This definition could be used in unknown dynamic systems: if every process in the system is an x-Anti-Source for $x \geq \lfloor \frac{n}{k+1} \rfloor + 1$, then Assumption 3.2 (with $\alpha = x$) and the quorum intersection property are ensured. However, the $M^{anti(x)}$ model only requires x processes to be x-Anti-Sources, which is only sufficient to implement quorums if $x = n$, since the intersection property must apply to every process in the system.

In [AGSS13], Arantes et al. present an algorithm that implements the Ω failure detector in an asynchronous TVG of class 5. To this end, the authors define the Stable Responsiveness Property ($\mathcal{SRP}$). A correct process p satisfies the $\mathcal{SRP}$ at time t if and only if, after t, all nodes in p's neighborhood receive a response from p to every one of their queries within the first α responses.

The definition of the $\mathcal{SRP}$ can be compared to the definition of an eventually winning process. Both properties enable a leader election mechanism by assuming that after some time, some process is among the first to respond to the queries of its neighbors. However, $\mathcal{SRP}$ applies to every process that shares a link with the leader after t, even for a moment, whereas the property of an eventually winning process p_l only applies to the processes of R_l, meaning those processes that interact infinitely often with p_l. Thus, a process can join the neighborhood of an eventually winning leader and leave it later on, which is not possible with a process satisfying the $\mathcal{SRP}$. But while the properties of an eventually winning leader can apply to a smaller subset of processes, those properties are stronger. Process p_l is not only required to respond to every query from its neighbors in time, it must also be the fastest to respond. Additionally, the processes within R_l are expected to communicate with each other to some extent, which is not necessary in the $\mathcal{SRP}$.

3.4 Failure Detector Algorithms

In this section a $\Sigma_{\perp,k}$ algorithm is first presented, then extended into a $Pi\Sigma_{\perp,x,y}$ algorithm.

3.4.1 An Algorithm for $\Sigma_{\perp,k}$

Algorithm 1 implements the $\Sigma_{\perp,k}$ failure detector in unknown dynamic systems with asynchronous communications. It uses a query/response mechanism with round numbers in order to ensure quorum liveness.

3.4.1.1 Assumptions

Algorithm 1 implements $\Sigma_{\perp,k}$ in unknown dynamic systems, provided that the following assumptions hold:

1. The system is a Time-Varying Graph of class 5-(α, γ) where α is the minimal size of a quorum and γ is the maximal time taken by a process to receive its own broadcasts (Assumption 3.1).

2. The run follows a generalized winning quorums message pattern (Assumption 3.2).

Algorithm 1 Implementation of $\Sigma_{\perp,k}$ for process p_i.

```
1:  init
2:      r_i ← 0 // Local round number
3:      qr_i ← ⊥ // The quorum returned by Σ_{⊥,k} for p_i
4:      recv_from_i ← {p_i} // Quorum buffer
5:      last_known_i ← ∅ // Round numbers of known processes
6:      bcast(p_i, 0, ∅)

7:  upon reception of (src, r_src, Q) from p_j do
8:      if src = p_i and r_src = r_i then // Response
9:          recv_from_i ← recv_from_i ∪ Q
10:         if |recv_from_i| ≥ α then
11:             qr_i ← recv_from_i
12:             recv_from_i ← {p_i}
13:             r_i ← r_i + 1
14:         bcast(p_i, r_i, ∅)
15:     else if src ≠ p_i then // Query
16:         if ∃last_r | ⟨src, last_r⟩ ∈ last_known_i
17:                 ∧ last_r ≤ r_src then
18:             last_known_i ← last_known_i \ {⟨src, last_r⟩}
19:             last_known_i ← last_known_i ∪ {⟨src, r_src⟩}
20:             bcast(src, r_src, Q ∪ {p_i})
21:         else if ⟨src, −⟩ ∉ last_known_i then
22:             last_known_i ← last_known_i ∪ {⟨src, r_src⟩}
23:             bcast(src, r_src, Q ∪ {p_i})
24:         else
25:             do nothing
```

3.4.1.2 Notations

Each process p_i uses the following local variables:

r_i is the local round number of process p_i.

qr_i is the quorum currently returned by the failure detector for process p_i.

$recv\_from_i$ is the quorum buffer, containing all the identities of the processes whose response has been received by p_i since the time it last formed a new (complete) quorum. When the buffer contains enough information (i.e., at least α process identities), it becomes the new quorum and $recv\_from_i$ is reinitialized.

$last\_known_i$ is the knowledge p_i has of other processes round numbers. This variable and the associated mechanisms are not necessary for the correctness of the algorithm, they are simply used to improve performance by limiting the number of useless transmitted messages.

Process p_i calls the bcast(src, r_src, Q) primitive to broadcast a message to the processes currently in its neighborhood. A message contains the following values:

src is the identity of the original sender of the query (which is not necessarily the immediate sender of the message, since queries are forwarded multiple times).

r_src is the round number of src when this query was issued. Process src ignores responses to previous rounds.

Q is the set of the identities of processes who responded to the current query. When the query goes back to process src, it will add the content of this set to its quorum buffer.

3.4.1.3 Algorithm Description

The principle behind the algorithm is the following: every process p_i keeps broadcasting queries for round r_i until it receives enough responses to form a quorum of size at least α, then it increments r_i and proceeds with the next round.

Contrarily to most query/response algorithms, Algorithm 1 only uses one type of messages. A message is both a query and a response, depending on which process receives it. Every message travels from process to process, until it goes back to the original message sender. If the test on line 8 is true, the message is considered as a response to the current round query. If instead the test on line 15 is true, the message is considered as a query from another process.

Every process identity received in a response for the current round is added to the $recv\_from_i$ buffer (line 9), and when the buffer size gets superior or equal to α, then a new quorum is formed by copying $recv\_from_i$ into qr_i and resetting the buffer (lines 10 – 13).

If a received message is a query from another process, p_i updates its local knowledge and then adds its own identity to the message and rebroadcasts it unless another query for a higher round has been previously received from the same emitter (lines 15 – 25).

At first glance it might look like process p_i only broadcasts its queries once (lines 6 and 14), but keep in mind that processes receive their own broadcasts. Therefore, after initially broadcasting a new query, p_i will receive it at most γ instants later and broadcast it again (line 14).

The same rebroadcasting approach applies for queries from other processes. Once p_i has received a message from src for round r_src, it will keep rebroadcasting it (lines 20 and 23) until it is informed that src moved on past round r_src (the test on lines 16 – 17).

Based on the assumption of generalized winning quorums, the only action necessary to ensure quorum intersection is to make sure that quorums are formed from at least α process identities, which is guaranteed by line 10.

Quorum liveness is ensured because (1) correct processes keep forming new quorums from fresh information infinitely often thanks to class 5-(α, γ) and (2) the identities of crashed processes are excluded from new quorums since the r_src in their responses are eventually outdated (line 8).

3.4.1.4 Proof of Correctness

Lemma 3.1. *In a TVG of class 5-(α, γ) where Assumption 3.2 holds, Algorithm 1 ensures the quorum intersection property of $\Sigma_{\perp,k}$.*

Proof. Assumption 3.2 implies that $\sum_{i=1}^{m} w_i \leq k$. For any number $w \in [1, k]$, n_w denotes the number of winning quorums of weight w. It follows that $\sum_{w=1}^{k} w \times n_w \leq k$.

Additionally, Assumption 3.2 imposes that every response set includes responses from a winning quorum Q_{wi} of weight w_i such that at least $\lfloor \frac{|Q_{wi}|}{w_i+1} \rfloor + 1$ processes from Q_{wi} are part of that response set. It follows that, if $w_i + 1$ response sets are formed from the same winning quorum Q_{wi}, at least two of these response sets intersect.

If no two response sets are to intersect, then at most w_i response sets can be formed from a given winning quorum Q_{wi}. Therefore, for any number $w \in [1, k]$, at most $w \times n_w$ response sets can be formed from the set of all winning quorums of weight w. It follows finally that at most $\sum_{w=1}^{k} w \times n_w$ response sets can be formed from the set of all winning quorums without any two of them intersecting. Since $\sum_{w=1}^{k} w \times n_w \leq k$, at least two out of any $k + 1$ response sets intersect.

Lines 10 and 11 of Algorithm 1 ensure that quorums include the first α responses (response set) to the current query. Therefore every quorum includes a response set, and the quorum intersection property of $\Sigma_{\perp,k}$ is ensured. □

Lemma 3.2. *In a TVG of class 5-(α, γ), every correct process executing Algorithm 1 forms a new quorum infinitely often.*

Proof. Since it uses a query-response mechanism, Algorithm 1 requires every correct process to reach and be reached back by α processes, which is ensured by a TVG of class 5-(α, γ) infinitely often. Even if a journey includes waiting time during which the process holding the message is isolated, the process keeps memory of the message by rebroadcasting it to itself, and transmits it to other processes as soon as it it stops being isolated. As a result, every correct process will receive responses from α processes infinitely often, and therefore pass the test on line 10 infinitely often. □

Lemma 3.3. *In a TVG of class 5-(α, γ), Algorithm 1 ensures the quorum liveness property of $\Sigma_{\perp,k}$.*

Proof. By definition, faulty processes will crash or leave the system forever in a finite time. Let $t \in \mathcal{T}$ be the time at which the last faulty process crashes or leaves the system forever. Since $f < n$, there are correct processes in the system. Lemma 3.2 ensures that each of these processes forms a new quorum sometime after t. Let $\tau \in \mathcal{T}$ be a time such that $\tau > t$ and every remaining process has formed a *quorum* between t and τ. Therefore, every quorum being currently built at τ has been started after t, which means no faulty process can possibly respond to the corresponding query message. As a result, every new quorum formed after τ contains only correct processes. It follows that Algorithm 1 ensures the quorum liveness property of $\Sigma_{\perp,k}$. □

Lemma 3.4. *In a TVG of class 5-(α, γ), Algorithm 1 ensures the quorum connectivity property of $\Sigma_{\perp,k}$.*

Proof. The properties of a TVG of class 5-(α, γ) ensure that every correct process will always receive enough messages to pass the test on line 10 and keep forming new quorums infinitely often. The test on line 8 ensures that processes only form quorums from messages from the current round. It follows that eventually, every correct process p_i only includes in its quorums processes which receive its queries and respond to it infinitely often. Therefore, p_i can send and receive messages infinitely often to and from the processes that are infinitely often in its quorums.

Let $p_i \in \mathcal{C}$ and $p_j \in R_i$. By definition of R_i, p_i and p_j's quorums intersect infinitely often and thus there must exist a correct process p_m such that p_m is infinitely often in p_i's quorums and p_m is infinitely often in p_j's quorums. As a result, p_m can receive messages from p_i infinitely often and p_j can receive messages from p_m infinitely often. Therefore if messages are routed through p_m, p_j can receive messages from p_i infinitely often. □

Theorem 3.1. *In a TVG of class 5-(α, γ) where Assumption 3.2 holds, Algorithm 1 implements a $\Sigma_{\perp,k}$ failure detector.*

Proof. It follows from Lemmas 3.1, 3.3 and 3.4 that the algorithm ensures the quorum intersection, quorum liveness and quorum connectivity properties respectively.

Self-inclusion is ensured by the fact that every quorum is formed from the buffer $recv\_from_i$ (line 11), and the buffer is always initialized with p_i (lines 4 and 12). □

3.4.2 An Algorithm for $\Pi\Sigma_{\perp,x}$

Algorithm 2 is an extension of Algorithm 1 aiming at implementing $\Pi\Sigma_{\perp,x}$ in unknown dynamic systems. It adds an election mechanism to the original algorithm in order to identify an eventual partial leader.

This leader election mechanism relies on the quorum order, as defined in Section 3.3.2.2. Every time a process forms a new quorum, it selects the first process in the quorum as candidate for the leader election. If a process is the candidate of every other process in its quorum, then it selects itself as leader; otherwise it selects its candidate as leader.

3.4.2.1 Assumptions

Algorithm 2 implements $\Pi\Sigma_{\perp,x}$ in unknown dynamic systems, provided that the following assumptions hold:

1. The system is a Time-Varying Graph of class 5-(α, γ) where α is the minimal size of a quorum and γ is the maximal time taken by a process to receive its own broadcasts (Assumption 3.1).
2. The run follows a generalized winning quorums message pattern (Assumption 3.2).
3. The system verifies the eventually winning γ-sources assumption (Assumption 3.3).

3.4.2.2 Notations

Algorithm 2 uses the same notations as Algorithm 1. Additionally, each process p_i uses the following local variables:

$leader_i$ is the leader returned by the failure detector for process p_i. $leader_i$ is initially p_i, and is later updated on lines 21 or 23.

Algorithm 2 Implementation of $\Pi\Sigma_{\perp,x}$ for process p_i.

```
1:  init
2:      r_i ← 0 // Local round number
3:      qr_i ← ⊥ // The quorum returned by ΠΣ_{⊥,x} for p_i
4:      recv_from_i ← {p_i} // Quorum buffer
5:      last_known_i ← ∅ // Round numbers of known processes
6:      leader_i ← p_i // The leader returned by ΠΣ_{⊥,x} for p_i
7:      candidate_i ← ⊥ // p_i's current candidate for leadership
8:      candidates_i ← ∅ // Candidates of processes in recv_from_i
9:      bcast(p_i, 0, ∅, ∅)

10: upon reception of (src, r_src, Q, cands) from p_j do
11:     if src = p_i and r_src = r_i then // Response
12:         recv_from_i ← recv_from_i ∪ Q
13:         candidates_i ← candidates_i ∪ cands
14:         if |recv_from_i| ≥ α then
15:             qr_i ← recv_from_i
16:             recv_from_i ← {p_i}
17:             r_i ← r_i + 1
18:             candidate_i ← p_l | (pos(p_l, qr_i) = 1 ∧ p_l ≠ p_i)
19:                 ∨(pos(p_i, qr_i) = 1 ∧ pos(p_l, qr_i) = 2)
20:             if candidates_i = {p_i} or ∅ then
21:                 leader_i ← p_i
22:             else
23:                 leader_i ← candidate_i
24:             candidates_i ← ∅
25:         bcast(p_i, r_i, ∅, ∅)
26:     else if src ≠ p_i then // Query
27:         if ∃last_r | ⟨src, last_r⟩ ∈ last_known_i
28:                 ∧ last_r ≤ r_src then
29:             last_known_i ← last_known_i \ {⟨src, last_r⟩}
30:             last_known_i ← last_known_i ∪ {⟨src, r_src⟩}
31:             bcast(src, r_src, Q ∪ {p_i}, cands ∪ {candidate_i})
32:         else if ⟨src, −⟩ ∉ last_known_i then
33:             last_known_i ← last_known_i ∪ {⟨src, r_src⟩}
34:             bcast(src, r_src, Q ∪ {p_i}, cands ∪ {candidate_i})
35:         else
36:             do nothing
```

$candidate_i$ is the first process in p_i's most recent quorum (excluding p_i itself). It is affected in lines 18 – 19. $candidate_i$ is initialized to $\perp$ and is added to sets (lines 31 and 34). As a convention, $\emptyset \cup \{\perp\} = \emptyset$.

$candidates_i$ is the set of the candidates of the processes in $recv\_from_i$ (except p_i). p_i will only elect itself as leader (line 21) if $candidates_i$ only contains p_i (i.e., p_i is the candidate of every process in $recv\_from_i \backslash \{p_i\}$) or if $candidates_i$ is empty (i.e., p_i considers itself alone).

In addition to the message parameters described for Algorithm 1, messages sent by processes contain the *cands* parameter, which is the set of the candidates of the processes in Q, at the time when they responded to the query. It carries the information necessary for process p_i to

build its $candidates_i$ set on line 13.

3.4.2.3 Algorithm Description

Algorithm 2 follows the same structure and uses the same mechanisms to build quorums as Algorithm 1. Its additional code aims to select partial leaders according to the eventual partial leadership property of $\Pi\Sigma_{\perp,x}$. The extension added to Algorithm 1 is composed of two parts: candidate selection and leader selection.

Candidate selection revolves around the notion of quorum order presented in Section 3.3.2.2. The first process in every quorum is selected as the candidate. Whenever a process p_i completes a new quorum (meaning it passes the test on line 14), it handles the end of the round similarly to Algorithm 1 (lines 15 – 17). It then identifies the first process in the new quorum (excluding itself) according to the chosen ordering method in lines 18 – 19 and selects it as its $candidate_i$. If it was possible for p_i to be its own candidate, and if quorums were ordered by date of response, then p_i would always be its own candidate.

By virtue of Assumption 3.3, an eventually winning process p_l will eventually be forever the candidate of every process in $R_l \backslash \{p_l\}$. However, p_l cannot be its own candidate. Therefore, information about p_l's own quorum order is not sufficient for p_l to select itself as the leader. It must take into account the candidates of other processes.

This is the purpose of the $candidates_i$ variable. Other processes inform p_i of their respective candidates by including it in their responses (lines 31 and 34), and p_i gathers this information in $candidates_i$ in line 13. When p_i completes a quorum, $candidates_i$ contains the candidates of the processes currently in $qr_i \backslash \{p_i\}$.

If every process in qr_i agrees on p_i as the candidate (or if p_i is the only process in qr_i), then p_i selects itself as the leader (line 21). Otherwise, p_i selects $candidate_i$ (line 23).

Note that point (3) of Definition 3.5 prevents the problematic case where a process p_i only includes in its quorums an eventually winning process p_l and processes in $\Pi \backslash R_l$. In this scenario, it would be possible for every process in R_i (including p_l) to chose p_i as its candidate, thus misleading p_i into selecting itself as the leader infinitely often.

3.4.2.4 Proof of Correctness

This proof will show that, if Assumptions 3.1, 3.2 and 3.3 hold, then Algorithm 2 ensures the 5 properties of $\Pi\Sigma_{\perp,x}$.

Lemma 3.5. *In a TVG of class 5-(α, γ) where Assumption 3.2 holds, Algorithm 2 ensures the self-inclusion, quorum intersection, quorum liveness and quorum connectivity properties of $\Pi\Sigma_{\perp,x}$.*

Proof. The added code from Algorithm 1 does not modify the way the qr_i variable is initialized and updated. Therefore, the proof for Theorem 3.1 holds for Algorithm 2. □

Lemma 3.6. *Every eventually winning process p_l is eventually forever the $candidate_i$ of every process $p_i (\neq p_l)$ of its recurrent neighborhood.* $\forall p_l \in W, \forall p_i \in R_l \backslash \{p_l\} : \exists \tau : \forall \tau' \geq \tau :$ *$candidate_i = p_l$ at time τ'.*

Proof. It follows from the properties of a TVG of class 5-(α, γ) that correct processes will keep passing the test on line 14, and therefore form new quorums infinitely often.

By contradiction, assume the following:

$$\exists p_l \in W, \exists p_i \in R_l \backslash \{p_l\}, \exists p_m \in \Pi \backslash \{p_l\}, \forall \tau : \exists \tau' \geq \tau : candidate_i = p_m \text{ at time } \tau'$$

There are, thus, two cases:

$\boldsymbol{p_m \notin R_i}$. By definition of R_i, there is a time after which p_i's quorums never intersect with p_m's quorum. By construction of the algorithm (lines 4 and 16), self-inclusion is ensured (every process belongs to its own quorums). Thus, there is a time after which p_m is never in p_i's quorums, and therefore it can never be selected as $candidate_i$ on lines 18 – 19 after this time.

$\boldsymbol{p_m \in R_i}$. Since p_l is an eventually winning process, there is a time after which (1) p_l is in every quorum formed by p_i and (2) in every quorum formed by p_i that includes p_m, p_l is positioned before p_m. As a result, p_m can never be selected as $candidate_i$ on lines 18 – 19 after this time. □

W is the set of all eventually winning processes, and L is the set of all eventual partial leaders.

Lemma 3.7. *Every eventually winning process is an eventual partial leader. $W \subseteq L$.*

Proof. Let $p_l \in W$. p_l is an eventual partial leader if and only if, for every $p_i \in R_l$, eventually $leader_i = p_l$ forever. There are two cases:

$\boldsymbol{p_i = p_l}$. It follows from the definition of R_l and from self-inclusion that there is a time after which every process that is not in R_l will stop appearing in the quorums formed by p_l. It follows that there is a time τ_1 such that $\forall \tau_1' > \tau_1, qr_l^{\tau_1'} \subseteq R_l$. If $\alpha = 1$, then $qr_l^{\tau_1'} = \{p_l\}$ and therefore $candidates_l = \emptyset$ at time τ_1' (by construction of $candidates_l$). If $\alpha > 1$, since the definition of R_l is symmetrical, $\forall \tau_1' > \tau_1, \forall p_j \in qr_l^{\tau_1'} : p_l \in R_j$. It then follows from Lemma 3.6 that $\exists \tau_2 \geq \tau_1, \forall \tau_2' > \tau_2, \forall p_j \neq p_l \in qr_l^{\tau_2'} : candidate_j = p_l$ at time τ_2'. Since p_l will keep forming new quorums with fresh information, $\exists \tau_3 \geq \tau_2$ such that every time after τ_3 that p_l completes a round, then $candidates_l = \{p_l\}$. As a result, after time τ_3, p_l will always pass the test on line 20 and, therefore, will forever identify itself as the leader.

$\boldsymbol{p_i \neq p_l}$. According to point (3) of the eventually winning process definition, $\exists \tau_1, \forall \tau_1' > \tau_1, \exists p_j \in R_l : p_j \in qr_i^{\tau_1'}$. It follows from Lemma 3.6 that $\exists \tau_2 \geq \tau_1, \forall \tau_2' > \tau_2, candidate_j = candidate_i = p_l$ at time τ_2'. Since p_i will keep forming new quorums with fresh information received from p_j, $\exists \tau_3 \geq \tau_2$ such that every time after τ_3 that p_l completes a round, then $p_l \in candidates_i$. As a result, after τ_3, p_i will always fail the test on line 20 and will forever identify $candidate_i = p_l$ as the leader.

In both cases, p_i selects p_l as leader forever, which makes p_l an eventual partial leader. □

Lemma 3.8. *If the eventually winning γ-sources assumption (Assumption 3.3) holds, then Algorithm 2 ensures the eventual partial leadership property of $\Pi\Sigma_{\perp,x}$.*

Proof. It follows from Assumption 3.3 that $\forall p_i \in \mathcal{C}, \exists p_l \in W, \forall \tau : \exists \mathcal{J} \in \mathcal{J}^{\gamma}_{(p_l,p_i)} \wedge departure(\mathcal{J}) > \tau$. It follows from Lemma 3.7 that $p_l \in L$. Since fair-lossy channels are assumed, then if p_l sends messages infinitely often, then p_i will receive messages from p_l infinitely often. □

Theorem 3.2. *In a TVG of class 5-(α, γ) where Assumptions 3.2 and 3.3 hold, Algorithm 2 implements a $\Pi\Sigma_{\perp,x}$ failure detector.*

Proof. Follows directly from Lemmas 3.5 and 3.8. □

3.4.3 An Algorithm for $\Pi\Sigma_{\perp,x,y}$

An algorithm for $\Pi\Sigma_{\perp,x,y}$ simply consists in executing y instances of Algorithm 2 simultaneously. This algorithm relies on Assumptions 3.1 , 3.2 and 3.3. However, Assumption 3.3 is only required to apply for one out of the y instances of the algorithm.

3.5 A k-Set Agreement Algorithm

In [MRS12], the authors proposed an algorithm for k-set agreement using $\Pi\Sigma_{x,y}$ for static networks. The k-set algorithm itself is very simple. It only deals with the liveness property of k-set agreement (termination) and encapsulates the safety properties (validity and agreement) into the $Alpha_x$ sub protocol. This section will adapt the $Alpha_x$ and k-set agreement algorithms for dynamic networks.

3.5.1 The $Alpha_x$ Sub Protocol

Alpha was introduced in [GR07] as a way to exactly capture the safety properties of consensus (that is, validity and agreement). It is thus complementary to the Ω failure detector, which is necessary to ensure liveness (the termination property). *Alpha* was later generalized in [RT06] into KA for the k-set agreement problem.

In [MRS12], Mostéfaoui, Raynal and Stainer define $Alpha_x$ as an extended, weaker version of the KA of [RT06]. $Alpha_x$ is a distributed object used to store values proposed by processes. It initially stores the default value $\perp$. It provides processes with an operation $Alpha.propose_x(r, v)$ that returns a value (possibly $\perp$). The round number r is a logical time and v is a proposed value. It is assumed that (a) each process will use increasing round numbers in successive invocations of $Alpha.propose_x()$ and (b) distinct processes use different round numbers. An $Alpha_x$ object is defined by the following properties:

- **Termination**: Any invocation of $Alpha.propose_x()$ by a correct process terminates.
- **Validity**: If $Alpha.propose_x(r, v)$ returns $v' \neq \perp$, then $Alpha.propose_x(r', v')$ has been invoked with $r' \leq r$.
- **Quasi-agreement**: At most x different non-$\perp$ values can be returned by different invocations of $Alpha.propose_x()$.
- **Obligation**: Let p_l be a correct process and $Q(l, \tau) = \{p_i \in \mathcal{C} | \forall \tau_i, \tau_l \geq \tau : qr_i^{\tau_i} \cap qr_l^{\tau_l} = \emptyset\}$. If, after time τ, (a) only p_l and processes of $Q(l, \tau)$ invoke $Alpha.propose_x()$ and (b) p_l invokes $Alpha.propose_x()$ infinitely often, then at least one invocation issued by p_l returns a non-$\perp$ value.

Note that the termination property of $Alpha_x$ is not related to the termination property of the k-set agreement.

In order to ensure the safety properties of k-set agreement, it is not necessary to make use of the eventual partial leadership property of $\Pi\Sigma_{\perp,x}$ and therefore, the $Alpha_x$ algorithm presented here does not make use of the $leader_i$ variable. However, the k-set agreement algorithm implements the termination property of k-set agreement by relying on the obligation property of $Alpha_x$ and the eventual partial leadership property of $\Pi\Sigma_{\perp,x}$.

The definitions in [BT10] and [MRS12], that propose k-set agreement algorithms for static networks, use different obligation properties. The $Alpha_x$ in this thesis is the one defined in [MRS12], which is weaker than the one in [BT10] by being Σ_x-aware.

3.5.2 $Alpha_x$ Algorithm

This section proposes an algorithm implementing $Alpha_x$ for unknown dynamic systems model enriched with $\Pi\Sigma_{\perp,x}$, adapted from the algorithm in [MRS12].

The algorithm gives each proposed value a priority. Each process p_i keeps a value est_i, which is its current estimation of the value it will decide, and a pair (lre_i, pos_i) which defines the priority of value est_i. lre_i is the highest round seen by p_i and pos_i is the position of value est_i within round lre_i. The position is used to fix priority on proposed values.

The function $g(\rho, \delta) = 2^\delta(\rho - 1) + 1$ where ρ is the position of value v on round r and $\delta = r' - r$, with $r' \geq r$, is used to compute the position of v on round r'.

If value v has priority ρ at round r and value v' has priority ρ' at round r' with $r \leq r'$, v has lower priority than v' at round r' if and only if $g(\rho, r' - r) < \rho'$ or $(g(\rho, r' - r) = \rho') \wedge (v < v')$.

The $Alpha.propose_x()$ function is composed of two phases. In the read phase (lines 4 – 9), the process attempts to gather knowledge on the values proposed by other processes in a quorum (as defined by $\Sigma_{\perp,x}$) by sending REQ_R messages and receiving RSP_R messages. If a process in the quorum is already computing a higher round, p_i returns $\perp$ (line 7). Otherwise, it selects the highest priority value it knows of (lines 8 – 8), and proceeds to the write phase.

In the write phase (lines 10 – 16), the process attempts to raise the priority of its current estimated value by communicating it to other processes in a quorum with REQ_W messages and receiving RSP_W messages. Once again, if any process in the quorum is computing a higher round, p_i returns $\perp$ (line 15). If another process has a value of higher priority for the current round, p_i adopts it as its new estimated value (lines 16 – 16). p_i then raises pos_i by 1 (line 11) and repeats the write phase until it manages to raise a value to position 2^r (line 10) or until it encounters a process in a higher round (line 15).

The following modifications were made to the original algorithm in [MRS12] in order for the algorithm to ensure the properties of $Alpha_x$ in dynamic networks:

The original algorithm assumed a complete, static communication graph with reliable channels and therefore every message was only sent once. In an unknown dynamic system, messages need to be rebroadcast (lines 22, 23, 31 and 33). This mechanism ensures that (1) the emitting process will rebroadcast its own message every γ units of time; and (2) the reception of the message will not be restricted to the neighbors of the emitting process. The message will be received by every process that can be reached through a γ-journey.

Since messages are rebroadcast, the direct emitter of a message is not necessarily the source of the message. For this reason, the process identifier of the responding process was added in message types RSP_R and RSP_W.

Algorithm 3 Implementation of $Alpha_x$ using $\Pi\Sigma_{\perp,x}$ in dynamic networks for process p_i.

```
1:  init
2:      lre_i ← 0; est_i ← ⊥; pos_i ← 0

3:  function ALPHA.PROPOSE_X(r,v_i)
4:      repeat Q_i ← qr_i; bcast REQ_R(r, Q_i)
5:      until Q_i ≠ ⊥ and ∀p_j ∈ Q_i : RSP_R(r, p_j, ⟨lre_j, pos_j, val_j⟩) received
6:      rcv_i ← {⟨lre_j, pos_j, est_j⟩ : p_j ∈ Q_i∧ RSP_R(r, p_j, ⟨lre_j, pos_j, est_j⟩) received}
7:      if ∃⟨lre, −, −⟩ ∈ rcv_i : lre > lre_i then return(⊥)
8:      pos_i ← max{pos | ⟨r, pos, v⟩ ∈ rcv_i}; est_i ← max{v | ⟨r, pos_i, v⟩ ∈ rcv_i}
9:      if est_i = ⊥ then est_i ← v_i
10:     while pos_i < 2^r do
11:         pos_i ← pos_i + 1; pst_i ← pos_i
12:         repeat Q_i ← qr_i; bcast REQ_W(r, pst_i, est_i, Q_i)
13:         until Q_i ≠ ⊥ and ∀p_j ∈ Q_i : RSP_W(r, pst_i, p_j, ⟨lre_j, pos_j, val_j⟩) received
14:         rcv_i = {⟨lre_j, pos_j, est_j⟩ : p_j ∈ Q_i∧ RSP_W(r, pst_i, p_j, ⟨lre_j, pos_j, est_j⟩) received}
15:         if ∃lre : ⟨lre, −, −⟩ ∈ rcv_i : lre > lre_i then return(⊥)
16:         pos_i ← max{pos | ⟨r, pos, v⟩ ∈ rcv_i}; est_i ← max{v | ⟨r, pos_i, v⟩ ∈ rcv_i}
17:     return(est_i)

18: upon reception of REQ_R(rd, Q) do
19:     if p_i ∈ Q then
20:         if rd > lre_i then pos_i ← g(pos_i, rd − lre_i); lre_i ← rd
21:         bcast RSP_R(rd, p_i, ⟨lre_i, pos_i, est_i⟩)
22:     bcast REQ_R(rd, Q)

23: upon reception of RSP_R(rd, p_j, ⟨lre_j, pos_j, est_j⟩) do bcast RSP_R(rd, p_j, ⟨lre_j, pos_j, est_j⟩)

24: upon reception of REQ_W(rd, pos, est, Q) do
25:     if p_i ∈ Q then
26:         if rd ≥ lre_i then
27:             pos_i ← g(pos_i, rd − lre_i); lre_i ← rd
28:             if pos > pos_i then est_i ← est; pos_i ← pos
29:             else if pos = pos_i then est_i ← max{est_i, est}
30:         bcast RSP_W(rd, pos, p_i, ⟨lre_i, pos_i, est_i⟩)
31:     bcast REQ_W(rd, pos, est, Q)

32: upon reception of RSP_W(rd, pos, p_j, ⟨lre_j, pos_j, est_j⟩) do
33:     bcast RSP_W(rd, pos, p_j, ⟨lre_j, pos_j, est_j⟩)
```

The original algorithm uses a selective multicast for both the read and write phases, i.e., messages are sent only to the processes in a quorum Q_i. This new algorithm uses broadcasts as defined in Section 3.1 (lines 4 and 12) and transmits Q_i with the message. All receiving processes will rebroadcast the message, but only the processes within Q_i will deliver it (lines 19 and 25).

Theorem 3.3. *In unknown dynamic systems augmented with $\Pi\Sigma_{\perp,x}$, Algorithm 3 ensures the*

properties of Alpha$_x$.

Proof. The modifications added to the original algorithms from [BT10] and [MRS12] do not allow the algorithm to add new values, therefore the proof for validity in the original papers holds. Similarly, the proofs for obligation in [MRS12] and quasi-agreement in [BT10] do not rely on any static connectivity assumption, and instead rely on algorithm behavioural properties which were not altered in this new version. Therefore, the original proofs hold for Algorithm 3.

Concerning termination, the only possibility for an invocation not to terminate is that process p_i waits forever for a response message in one of the repeat loops (lines 4–5 and 12–13). Assume by contradiction that p_i waits forever for responses. The liveness property of $\Pi\Sigma_{\perp,x}$ ensures that eventually p_i only sends queries to correct processes and waits for responses from correct processes. Given that the set of correct processes is finite, the set of possible correct quorums is finite too. It follows that there is a correct quorum Q such that infinitely often, $qr_i = Q$, and therefore according to the quorum connectivity and self-inclusion properties of $\Pi\Sigma_{\perp,x}$, there are recurrent journeys between any process in Q and p_i. As a result, all the processes from Q will eventually receive the queries from p_i, and p_i will eventually receive the responses from the processes in Q, and, therefore, exit the repeat loop. □

3.5.3 k-Set Agreement Algorithm

Given an *Alpha*$_x$ object and a $\Pi\Sigma_{\perp,x,y}$ failure detector, solving k-set agreement is simple. The algorithm given here is an adaptation of the one given in [MRS12] for dynamic networks. x-Set agreement is first solved with $\Pi\Sigma_{\perp,x}$ (Algorithm 4), and then k-set agreement with $\Pi\Sigma_{\perp,x,y}$ for $k \geq xy$.

Algorithm 4 x-Set agreement with *Alpha*$_x$ using $\Pi\Sigma_{\perp,x}$ in dynamic networks for process p_i.

```
1:  init
2:      dec_i ← ⊥ // The value decided by p_i (⊥ if p_i has not decided)
3:      prime_i ← the i^th prime number // Constant
4:      r_i ← prime_i // The current round number

5:  function PROPOSE(v_i)
6:      while dec_i = ⊥ do
7:          if leader_i = p_i then
8:              dec_i ← Alpha.propose_x(r_i, v_i)
9:              r_i ← r_i × prime_i
10:     decide(dec_i)
11:     bcast DECISION(dec_i)

12: upon reception of DECISION(d) do
13:     if dec_i = ⊥ then
14:         dec_i ← d
15:         decide(d)
16:     bcast DECISION(d)
```

A well formed invocation of *Alpha.propose*$_x(r, v)$ is an invocation such that two processes cannot use the same round number r, and successive round numbers for a given process are

increasing. To this end, each process p_i initially computes $prime_i$, the i^{th} prime number. p_i then uses $prime_i$ as its first round number, and multiplies it by $prime_i$ after every round. As a result, the round number of p_i increases and is always a power of $prime_i$, which ensures that two distinct processes always use distinct round numbers.

Theorem 3.4. *In unknown dynamic systems augmented with $\Pi\Sigma_{\perp,x}$ and with an $Alpha_x$ object, Algorithm 4 solves the x-set agreement problem.*

Proof. The test on line 6 ensures that the $\perp$ value is never decided. From this point on, the validity of the $Alpha_x$ object is enough to ensure the validity of x-set agreement. Similarly, the quasi-agreement property of $Alpha_x$ is enough to ensure the agreement property of x-set agreement.

The eventual partial leadership property of $\Pi\Sigma_{\perp,x}$ ensures that if every leader in L decides, then eventually every correct process will receive a DECISION message from a process in L. As a result, the proof for the termination property provided in [MRS12] holds for Algorithm 4. □

Similarly to [MRS12], a simple k-set algorithm can be obtained by running y instances of Algorithm 4, the j^{th} one ($1 \leq j \leq y$) relying on the component $FD_i[j]$ of failure detector $\Pi\Sigma_{\perp,x,y}$ for every process p_i. A process decides the same value decided by the first of the y instances that terminates. As there are y instances of the algorithm and each of them can decide x values at most, it follows that at most xy values can be decided. Therefore, the algorithm solves k-set agreement for $k \geq xy$.

3.6 Conclusion

This chapter adapted the existing $\Pi\Sigma_{x,y}$ failure detector to unknown dynamic systems by using the $\perp$ default value to deal with missing information and by adding connectivity properties to the failure detector definition. The result is the $\Pi\Sigma_{\perp,x,y}$ failure detector, which is sufficient to solve k-set agreement in unknown dynamic systems with $k \geq xy$.

An algorithm implementing $\Pi\Sigma_{\perp,x,y}$ in a Time-Varying Graph of class 5-(α, γ), relying on connectivity and message pattern assumptions, was presented.

Finally, an existing algorithm to solve k-set agreement was adapted for unknown dynamic systems augmented with $\Pi\Sigma_{\perp,x,y}$ ($k \geq xy$).

Chapter 4

The Weakest Failure Detector for Mutual Exclusion in Unknown Dynamic Systems

Contents

4.1 Model and Problem Definition 58
4.1.1 System Model 58
4.1.2 Failure Model 59
4.1.3 Connectivity Model 59
4.1.4 Knowledge Model 60
4.1.5 Problem Definition 60
4.2 Failure Detectors for Mutual Exclusion in Unknown Dynamic Systems 61
4.2.1 The $\mathcal{T}\Sigma^{l}$ Failure Detector 61
4.2.2 The $\mathcal{T}\Sigma^{lr}$ Failure Detector 61
4.3 Sufficiency of $\mathcal{T}\Sigma^{lr}$ to solve Fault-Tolerant Mutual Exclusion 62
4.3.1 Algorithm Description 62
4.3.2 Proof of Correctness 65
4.4 Necessity of $\mathcal{T}\Sigma^{lr}$ to solve Fault-Tolerant Mutual Exclusion 70

As discussed in Section 2.2.2, the majority of existing FTME algorithms consider static and known distributed systems without recovery. Hence, the conception of mutual exclusion in unknown dynamic distributed systems where crashed processes can recover presents great challenges.

A definition of recoverable mutual exclusion (RME) for systems with crash-recovery was presented in [GR16] and further studied in [GH17] and [JJ17]. A main change with regard to previous definitions of fault-tolerant mutual exclusion is the *critical section re-entry* property, which specifies that if a process p crashes while in the critical section and later recovers, then no

other process may enter the critical section until p re-enters it after its recovery. Intuitively, this means that the lock on the critical section is not released in the case of a temporary crash.

This chapter considers RME on top of a message passing model, where each process has access to a volatile memory of unbounded size, which is lost after a crash and recovery, and a non-volatile memory (stable storage) of bounded size. This model is called the *partial memory loss* model.

In [DGFGK05], the $\mathcal{T}$ failure detector was shown to be the weakest failure detector to solve fault-tolerant mutual exclusion in message passing systems with a majority of correct processes (see Section 2.3.4.1). Then, in [BCJ09], the $(\mathcal{T}, \Sigma^l)$ failure detector was shown to be the weakest failure detector to solve the same problem with no assumption on the number of process failures (see Section 2.3.4.2).

Both of these results are restricted to known, static systems without recovery. This chapter aims to extend these results to unknown dynamic systems where crashed processes can recover, with partial memory loss.

$(\mathcal{T}, \Sigma^l)$ is the sum of two failure detectors. It provides two outputs, one of which verifies the properties of $\mathcal{T}$, and another one which verifies the properties of Σ^l. Let $\mathcal{T}\Sigma^l$ be the detector which provides a single output verifying the properties of both $\mathcal{T}$ and Σ^l.

The contributions of this chapter are as follows:

1. A proof that $(\mathcal{T}, \Sigma^l)$ is equivalent to $\mathcal{T}\Sigma^l$ (Section 4.2);

2. The definition of the $\mathcal{T}\Sigma^{lr}$ failure detector, which is equivalent to $\mathcal{T}\Sigma^l$ in known, static systems without recovery, and is the weakest failure detector to solve RME in unknown dynamic systems (Section 4.2);

3. A RME algorithm that runs on top of the proposed model using the $\mathcal{T}\Sigma^{lr}$ failure detector and which tolerates crashes and recovery of processes, thus proving that $\mathcal{T}\Sigma^{lr}$ is sufficient to solve RME in unknown dynamic systems (Section 4.3);

4. A reduction algorithm proving the necessity of $\mathcal{T}\Sigma^{lr}$ to solve RME in unknown dynamic systems (Section 4.4).

4.1 Model and Problem Definition

This section presents the distributed system model used throughout the rest of the chapter and the definition of the Recoverable Mutual Exclusion (RME) problem.

4.1.1 System Model

The system is composed of a finite set of processes, denoted Π. Each process is uniquely identified. Additionally, *processes are asynchronous* (there is no bound on the relative speed of processes). They communicate by sending each other messages with a point-to-point SEND/RECEIVE primitive.

Communications are asynchronous (there is no bound on message transfer delay).

4.1.2 Failure Model

A process can *crash* (stop executing) during the run, and may *recover* from the crash, or not. A process can begin the run crashed, and start participating later in the run by recovering.

Each process has access to both a volatile memory and a stable storage of bounded size. After a crash and recovery, the variables in volatile memory are reset to their initial default values. As each process has access to stable storage, this model is called the *partial memory loss* model. In the rest of the chapter, the names of variables in stable storage is underlined.

A process is said to be *alive* at time t if it never stopped executing before t or if it recovered since the last time it stopped executing. A process which is not alive at time t is said to be *crashed* at time t.

In the traditional crash failure model, processes are grouped into *faulty* processes, which eventually crash, and *correct* processes, which never crash. However, in a crash-recovery model, in any run, three types of processes are considered[ACT00]:

1. *Eventually up* processes, which stop crashing after some time and remain alive forever. This type also includes processes that never crash (*always up*).

2. *Eventually down* processes, which eventually crash and never recover. This type also includes processes that crashed immediately at the start of the run and never recovered (*always down*).

3. *Unstable* processes, which crash and recover infinitely often. It is assumed that, infinitely often, each unstable process manages to stay alive long enough to at least send a message to each other process of which it is aware.

4.1.3 Connectivity Model

The system is *dynamic* in the sense that the edges in the communication graph can appear and disappear during the run. In other words, at any given time instant, each edge in the graph might or might not be available. Without any further assumption, a system in which no edge is ever available would fit this model. Since nothing can be computed in such a system, additional assumptions are needed. Therefore, the following properties are assumed to be verified:

- **Dynamic connectivity**: Every message sent by a process that is not *eventually down* to a process that is not *eventually down* is received at least once.

- **Uniqueness of reception**: Every message sent is received at most once.

- **First in, first out**: If process p sends a message m_1 to q and then sends m_2 to q, if q receives m_2 then it received m_1 first.

These properties imply not only that channels are reliable, but also that each pair of processes that are not *eventually down* is connected infinitely often by a path over time. This means that when a process p sends a message to process q, then there is a path from p to q such that at some point in the future, every edge on this path will be available in the correct order, and sufficiently long for the message to cross the edge. Note that it is not necessary that all the

edges on the path to be available at the same time, and the path that a pair of processes uses to communicate is not required to be the same every time. This connectivity assumption is referred to as a Time-Varying Graph of class $\mathcal{C}_5$ in [CFQS12].

The algorithms presented in this chapter assume that the underlying SEND/RECEIVE implementation handles message forwarding and, therefore, behaves the same way that it would in a complete communication graph with reliable channels.

4.1.4 Knowledge Model

The system is *unknown*, i.e., processes initially have no information on system membership or the number of processes of the system, and are only aware of their own identity. The identities of other processes can only be learned through exchanging messages. More practically, each process p has access to a local variable $\underline{known}_p$ (in stable storage) that initially contains only p. Eventually, $\underline{known}_p$ contains the set of all processes that are not eventually down. For the sake of simplicity, the algorithms in this chapter do not attempt to define the $\underline{known}_p$ variable and simply assume that an underlying discovery algorithm eventually fills it with the necessary process identities. This is not a strong assumption, since the *dynamic connectivity* property ensures that all processes will be able to communicate (and therefore learn of each other's existence) infinitely often.

4.1.5 Problem Definition

This chapter considers the Recoverable Mutual Exclusion (RME) problem, which is defined as follows. At any point in time, a process is either in the remainder, try, critical, or exit section. Every user is well-formed, that is that a user will go through the remainder, try, critical and exit sections in the correct order. In case of a crash and recovery, a well-formed user will restart in the critical section if it was in the critical section when it crashed, and will restart in the remainder section, otherwise (the *critical section re-entry* property of [GH17]).

A RME algorithm must provide a TRY SECTION and an EXIT SECTION procedures such that the following properties are satisfied:

- **Safety**: Two distinct alive processes p and q can not be in CS at the same time.
- **Liveness**: If an eventually up process p stopped crashing and is in the try section, then at some time later some process that is not eventually down is in CS.

Additionally, the following fairness property is considered:

- **Starvation Freedom**: If no process stays in its critical section forever, then every eventually up process that stopped crashing and reaches its try section will eventually enter its CS.

Note that stable storage is necessary to solve this problem. Indeed, if a process p is in the critical section when all processes simultaneously crash, without stable storage there is no way for p to re-enter critical section after recovery since no process in the system remembers that p was in the critical section in the first place.

4.2 Failure Detectors for Mutual Exclusion in Unknown Dynamic Systems

4.2.1 The $\mathcal{T}\Sigma^l$ Failure Detector

$(\mathcal{T}, \Sigma^l)$ provides two outputs, tr_p and qr_p, with tr_p verifying the properties of $\mathcal{T}$ and qr_p verifying the properties of Σ^l. $\mathcal{T}\Sigma^l$ is the failure detector that provides a single output tq_p verifying the properties of both $\mathcal{T}$ and Σ^l.

Theorem 4.1. *$(\mathcal{T}, \Sigma^l)$ is equivalent to $\mathcal{T}\Sigma^l$.*

Proof. $(\mathcal{T}, \Sigma^l)$ can be implemented using the output of $\mathcal{T}\Sigma^l$: it suffices to always return the value of tq_p as both tr_p and qr_p. Therefore, $(\mathcal{T}, \Sigma^l)$ is weaker than $\mathcal{T}\Sigma^l$.

$(\mathcal{T}, \Sigma^l)$ is sufficient to solve mutual exclusion, as shown in [BCJ09]. Section 4.4 will prove that it is possible to use RME to implement the $\mathcal{T}\Sigma^{lr}$ failure detector (defined in Section 4.2.2). The same reasoning and algorithms that are used in Section 4.4 can be used to show that it is possible to use FTME to implement $\mathcal{T}\Sigma^l$. It follows that $(\mathcal{T}, \Sigma^l)$ is sufficient to implement $\mathcal{T}\Sigma^l$, and as a result, $\mathcal{T}\Sigma^l$ is weaker than $(\mathcal{T}, \Sigma^l)$. □

4.2.2 The $\mathcal{T}\Sigma^{lr}$ Failure Detector

The existing definition of $\mathcal{T}\Sigma^l$ is for static, known networks, and therefore new definitions, suitable for unknown dynamic networks, must be provided.

In an unknown system, the lack of initial information renders difficult the implementation of some failure detector properties which must apply from the start of the run, in particular the intersection property. To circumvent this problem, the $\perp$ concept introduced in [RAS15] is used.

Additionally, the traditional properties of $\mathcal{T}\Sigma^l$ are expressed in terms of *correct* and *faulty* processes. $\mathcal{T}\Sigma^{lr}$ was rewritten using the concepts of *eventually up* and *eventually down* processes instead.

The $\mathcal{T}\Sigma^{lr}$ failure detector provides each process p with a set of trusted process identities, denoted tq_p, and a flag denoted rdy_p. rdy_p is initially set to $\perp$ and changes to $\top$ once the failure detector has gathered enough information to verify the live pairs intersection property. tq_p^t denotes the value of tq_p at time t, and rdy_p^t the value of rdy_p at time t. Process p is said to trust process q at time t if $q \in tq_p^t$, p is said to suspect q at time t if $q \notin tq_p^t$, and process p is said to be ready at time t if $rdy_p^t = \top$. The following properties must be verified.

- **Eventually strong accuracy**: Every *eventually up* process p is eventually trusted forever by every process that is not *eventually down*.
- **Strong completeness**: Every *eventually down* process p is eventually suspected forever by every process that is not *eventually down*.
- **Trusting accuracy**: For any process p, if there exist times t and $t' > t$ such that $q \in tq_p^t$ and $q \notin tq_p^{t'}$, then q is *eventually down* and will never be alive after t'.
- **Quorum readiness**: Every *eventually up* process is eventually ready forever.

- **Live pairs intersection**: If two processes p and q are both alive at time t, then for any couple of time instants $t_1 \leq t$ and $t_2 \leq t$, $(rdy_p^{t_1} = \top \wedge rdy_q^{t_2} = \top) \implies tq_p^{t_1} \cap tq_q^{t_2} \neq \emptyset$.

The eventually strong accuracy, strong completeness and trusting accuracy properties are the original properties of $\mathcal{T}$, adapted for a crash-recovery model. These properties are referred to as the trusting properties of $\mathcal{T}\Sigma^{lr}$.

Similarly, the strong completeness and live pairs intersection properties are the original properties of Σ^l, adapted for unknown dynamic systems. The new quorum readiness property, along with the rdy_p output variable, was added to deal with the lack of initial information in an unknown system. These properties are referred to as the quorum properties of $\mathcal{T}\Sigma^{lr}$.

Note that the strong completeness is both a trusting property and a quorum property, since both $\mathcal{T}$ and Σ^l make use of this same property.

Both trusting and quorum properties apply to the same set tq_p, which is different from preexisting definitions in which $\mathcal{T}$ and Σ^l are two separate oracles with separate outputs. Section 4.4, will prove that this combined version of the detector is necessary to solve RME.

Note that in a static, known system with reliable channels and prone to crash failures without recovery, $\mathcal{T}\Sigma^{lr}$ is equivalent to $\mathcal{T}\Sigma^l$, and therefore $(\mathcal{T}, \Sigma^l)$.

4.3 Sufficiency of $\mathcal{T}\Sigma^{lr}$ to solve Fault-Tolerant Mutual Exclusion

This section introduces Algorithm 5 and proves that it solves the RME in any unknown dynamic environment enriched with the $\mathcal{T}\Sigma^{lr}$ failure detector.

4.3.1 Algorithm Description

Let's consider p the sender (source) of a message. The following types of messages are used by Algorithm 5:

REQUEST: p has asked for permission to enter CS. The message contains the round number of the sender.

GRANT: p has granted permission to a requesting process to enter CS.

DONE: notifies other processes that p has just exited its CS.

REJECT: warns that p has already given its permission to another process different from itself, thus preventing deadlocks.

COMEBACK: notifies processes that p has just recovered from a crash.

UPDATE: p gives information to a recently recovered process q about p's requesting state, previously given permissions that p granted to q and vice-versa, and q's last round number of which p is aware.

In Algorithm 5, each request to enter the CS, issued by p, is tagged by a sequence round number.

Besides having access to the output, tq_p and rdy_p, of its local failure detector, process p also keeps the following local variables, initialized with the indicated value:

$\underline{crit}_p \leftarrow false$: a flag indicating that p is currently in CS. It is the only variable kept in stable storage. Thus, $\underline{crit}_p$ is not reinitialized after a crash and recovery.

Algorithm 5 Solving RME with $\mathcal{T}\Sigma^{lr}$: code for process p

1: **procedure** TRY SECTION
2: **wait for** $recovering_p = false$
3: $req_p \leftarrow true$
4: $round_p \leftarrow round_p + 1$; $grants_p \leftarrow \{p\}$
5: **for** $\forall q \in tq_p$ **do** SEND(REQUEST, $round_p, q$)
6: $requests_p \leftarrow requests_p \cup \{(round_p, p)\}$
7: CHECK REQUESTS()
8: **wait for** $gid_p = p$ **and** $rdy_p = \top$ **and** $tq_p \subseteq grants_p$
9: $\underline{crit}_p \leftarrow true$; $req_p \leftarrow false$
10: **procedure** EXIT SECTION
11: **wait for** $recovering_p = false$
12: $\underline{crit}_p \leftarrow false$
13: **for** $\forall q \in grants_p \setminus \{p\}$ **do** SEND(DONE, q)
14: $grants_p \leftarrow \{p\}$; $requests_p \leftarrow requests_p \setminus \{(*, p)\}$
15: CHECK REQUESTS()
16: **procedure** CHECK REQUESTS
17: **if** $(gid_p = -1$ **or** $gid_p = p)$ **and** $requests_p \neq \emptyset$ **and** $\underline{crit}_p = false$ **and** $recovering_p = false$ **then**
18: $(grnd_p, gid_p) \leftarrow$ HIGHEST($requests_p$)
19: **if** $gid_p \neq p$ **then** SEND(GRANT, gid_p)
20: **for** $\forall q \in grants_p \setminus \{p\}$ **do**
21: $grants_p \leftarrow grants_p \setminus \{q\}$
22: SEND(REJECT, q)
23: **procedure** RECONNECTION
24: $recovering_p \leftarrow true$
25: $update_p \leftarrow tq_p$
26: **for** $\forall q \in update_p$ **do**
27: SEND(COMEBACK, $\underline{crit}_p, q$)
28: **wait for** $update_p = \emptyset$
29: $recovering_p \leftarrow false$
30: CHECK REQUESTS()
31: **when** q added to tq_p
32: **if** $req_p = true$ **then** SEND(REQUEST, $round_p, q$)
33: **when** q removed from tq_p
34: $grants_p \leftarrow grants_p \setminus \{q\}$
35: $requests_p \leftarrow requests_p \setminus \{(*, q)\}$
36: $update_p \leftarrow update_p \setminus \{q\}$
37: **if** $gid_p = q$ **then**
38: $(gid_p, grnd_p) \leftarrow (-1, -1)$
39: CHECK REQUESTS()
40: **upon reception of** REQUEST ($round$) **from** src **do**
41: $requests_p \leftarrow requests_p \cup \{(round, src)\}$
42: $last\_round_p[src] \leftarrow round$
43: CHECK REQUESTS()
44: **upon reception of** GRANT () **from** src **do**
45: **if** $gid_p \neq -1$ **and** $gid_p \neq p$ **then**
46: SEND(REJECT, src)
47: **else if** $recovering_p = false$ **then**
48: $grants_p \leftarrow grants_p \cup \{src\}$
49: **upon reception of** DONE () **from** src **do**
50: $requests_p \leftarrow requests_p \setminus \{(*, src)\}$
51: $(gid_p, grnd_p) \leftarrow (-1, -1)$
52: CHECK REQUESTS()
53: **upon reception of** REJECT () **from** src **do**
54: $(gid_p, grnd_p) \leftarrow (-1, -1)$
55: CHECK REQUESTS()
56: **upon reception of** COMEBACK ($crit\_src$) **from** src **do**
57: $requests_p \leftarrow requests_p \setminus \{(*, src)\}$
58: **if** $crit\_src = false$ **and** $gid_p = src$ **then**
59: $(gid_p, grnd_p) \leftarrow (-1, -1)$
60: CHECK REQUESTS()
61: SEND(UPDATE, $gid_p = src, last\_round_p[src], src \in grants_p, round_p, req_p, src$)
62: **upon reception of** UPDATE ($grant\_p, last\_rnd, grant\_src, round, req$) **from** src **do**
63: $last\_round_p[src] \leftarrow round$
64: $round_p \leftarrow$MAX($round_p, last\_rnd$)
65: **if** $grant\_src = true$ **then** ▷ p previously granted src
66: $(gid_p, grnd_p) \leftarrow (src, round)$
67: **if** $grant\_p = true$ **then** ▷ src previously granted p
68: $grants_p \leftarrow grants_p \cup \{src\}$
69: **if** $req = true$ **then** ▷ src is requesting
70: $requests_p \leftarrow requests_p \cup \{(round, src)\}$
71: $update_p \leftarrow update_p \setminus \{src\}$

$round_p \leftarrow 0$: the local round number of p, which is used to number its requests. It is also used to define the current priority of p to access the critical section.

$last\_round_p \leftarrow \emptyset$: a table associating each known process identity with its last known round number. It is used to restore the round number of other processes after they crash and recover.

$req_p \leftarrow false$: a flag indicating that p is currently in the try section.

$requests_p \leftarrow \emptyset$: the set of requests received by p which are pending. Each request is a couple $(round, pid)$.

$gid_p \leftarrow -1$: the identity of the last process to which p granted its permission, or -1 if p did not grant it. It indicates that p sent a GRANT message to gid_p, and that this permission was not canceled by the reception of a DONE or REJECT message yet.

$grnd_p \leftarrow -1$: the current round number of the process to which p granted its permission, or -1 if p did not grant it.

$grants_p \leftarrow \{p\}$: the set of processes from which p received a GRANT message.

$recovering_p \leftarrow false$: a flag indicating that p is currently attempting to rebuild its volatile memory after a crash. Calls to TRY SECTION and EXIT SECTION will be delayed while $recovering_p = false$.

$update_p \leftarrow \emptyset$: the set of processes from which p waits for an UPDATE message. This variable is only used during the recovery phase, i.e., while $recovering_p = true$.

All of these local variables, except for $\underline{crit}_p$, are stored in volatile memory. This means that after a crash and recovery, they are reinitialized to the above default value.

Requests are totally ordered by their priority, which is defined as follows: $priority(round_p, p) > priority(round_q, q) \Leftrightarrow round_p < round_q$ **or** $[round_p = round_q$ **and** $p < q]$. The HIGHEST function takes a list of requests and returns the couple $(round, id)$ of the request with the highest priority among the trusted processes according to tq_p.

The CHECK REQUESTS procedure is extensively used in Algorithm 5. Provided that process p did not already grant its permission to another process and is not in CS, CHECK REQUESTS compares the requests that p received so far by calling the HIGHEST function (line 18), and sends a GRANT message to the process with the highest priority (line 19). In case p received grants from other processes before granting its own permission, it will send REJECT messages to the processes in $grants_p$ in order to prevent a deadlock (lines 20 – 22).

Whenever process p wants to access the critical section, it executes the TRY SECTION: p increments its $round_p$, resets its $grants_p$ set (line 4), and then broadcasts a REQUEST to every process in tq_p (line 5). If p gets knowledge of a new process while it is still in the try section, the request will also be sent to this process (line 32). Process p adds its own request to its $requests_p$ before calling CHECK REQUESTS (lines 6 – 7), and finally waits for permissions from every process in tq_p (and its own permission, line 8) before entering CS.

Upon reception of a REQUEST message from process q (lines 40 – 43), process p updates its knowledge about q's round number and adds the new request to its $requests_p$ set. It then calls CHECK REQUESTS to decide if it should send a grant to the new requester.

When receiving a GRANT message from process q, if p already granted its permission to some other process then it informs q by responding with a REJECT message to prevent deadlocks (line 46). Otherwise, if p is not in the recovery phase, it accepts q's permission by adding it to its $grants_p$ set.

Upon finishing the critical section and calling EXIT SECTION, p sends to all trusted processes a DONE message (line 13). Then, p resets its $grants_p$ set and cancels its request (line 50) before calling CHECK REQUESTS to grant its permission to the next process.

If p receives a DONE or REJECT message from process gid_p, it cancels the permission granted to gid_p (lines 51 and 54) and calls CHECK REQUESTS. In the case of a DONE message, the request from gid_p is also deleted from $requests_p$ (line 50), since gid_p is not requesting CS anymore. However, in the case of a REJECT, the request from gid_p is still valid and must be kept, even if it is not the highest priority request.

If p crashes and recovers, the RECONNECTION procedure will be called first. This procedure initiates the recovery phase (lines 24 – 29) by switching the $recovering_p$ flag to $true$, which will temporarily prevent the algorithm from going into the try or exit sections (lines 2 and 11) and from sending or accepting a grant (lines 17 and 47). During the recovery phase, p attempts to recover the information it lost during the crash by sending a COMEBACK message to every process in tq_p. Other processes will send UPDATE messages in response, which enables p to restore its $last\_round_p$, $round_p$, gid_p, $grnd_p$, and $requests_p$ variables (lines 63 – 71). The recovery phase ends when every process to which p sent a COMEBACK has either responded with an UPDATE message (line 71), or crashed (line 36). After recovering, p calls CHECK REQUESTS in order to choose a process to which it will grant its permission (line 30).

If p receives a COMEBACK message from a process q, it cancels any request previously received from q, since a process in recovery phase can only be in the remainder or critical section (by definition of a well-formed process). If q is in its remainder section ($\underline{crit}_p = false$), then p cancels any permission it might have granted to q previously (lines 58 – 60). Finally, p sends an UPDATE message to q.

Whenever p is informed by the failure detector that a process q is eventually down (lines 33 – 39), p deletes q from its $requests_p$, $grants_p$ and $update_p$ sets. If q was the process to which p granted permission, then p cancels the permission (line 38) and calls CHECK REQUESTS to grant its permission to another process, if appropriate.

4.3.2 Proof of Correctness

This section will prove, through the following claims, that any run of Algorithm 5 solves the RME problem.

Claim 4.1 (Safety)**.** *Two distinct alive processes p and q can not be in CS at the same time.*

In order to prove Claim 4.1, the following lemmata are required.

Lemma 4.1 (Uniqueness of the permission)**.** *Let p, q_1, q_2 be three distinct alive processes. If $p \in grants_{q_1}$ at a time t then p cannot send a* GRANT *message to q_2 at time t.*

Proof. The only way that p can send a GRANT message to a process q is on line 19, after it selected q as its gid_p. Note that the definition of the HIGHEST function also implies that $q \in tq_p$ at the time when the GRANT message is sent.

Suppose that p has sent a GRANT message at time t_G to another process q_1 (and therefore at time t_G, $gid_p = q_1$).

Assume that there is a time $t > t_G$ such that $p \in grants_{q_1}$. Then suppose that p sends a GRANT message to another process q_2 at time t.

In order to send a GRANT message to q_2, p has to set gid_p to -1 or to p at some time $t' \in [t_G, t]$ (otherwise p cannot pass the test on line 17). This affectation can only be done in one of the following lines:

Line 38: then $q_1 \notin tq_p^{t'}$. Since $q_1 \in tq_p^{t_G}$, according to the trusting accuracy property of $\mathcal{T}\Sigma^{lr}$, q_1 has crashed at some time before t' and will never recover. It is therefore impossible that $p \in grants_{q_1}$ at time t.

By resetting gid_p to -1 after a crash. If p crashed between t_G and t', then its gid_p got reset to -1. This also means that p entered the recovery phase (lines 24 – 29) at some time $t'' \in [t_G, t']$. Since $q_1 \in tq_p^{t_G}$, then according to the trusting accuracy property of $\mathcal{T}\Sigma^{lr}$, either q_1 crashed before t'' and will never recover (which is a contradiction), or $q_1 \in tq_p^{t''}$. p will therefore send a COMEBACK message to q_1 on line 27, and q_1 will respond with a UPDATE message with the $grant\_src$ parameter set to $true$, which will cause p to set its gid_p back to q_1. Since p cannot have sent a GRANT message while in the recovery phase (because of the test on line 17), then p cannot send the GRANT to q_2 at time t which is a contradiction.

Line 59: then p received a COMEBACK message from q_1 at some time $t'' \in [t_G, t']$. This means that q_1 crashed and went into the recovery phase. p will respond with an update message to q_1. Since q_1 cannot leave the recovery phase until it receives p's update and because of the first in, first out property, then p's GRANT message to q_1 was received either (1) before q_1 crashed, in which case the GRANT was forgotten, or (2) during the recovery phase, in which case q_1 will ignore the GRANT because of the test on line 47. In both cases, $p \notin grants_{q_1}$ after t'', which is a contradiction.

Line 51 or 54: then p received a DONE or REJECT message from q_1 at time t'. There are two cases. If q_1 sent the DONE or REJECT message after receiving the GRANT, then q_1 removed p from $grants_{q_1}$ on line 14 (resp. line 21) and did not add it back in afterwards, which is a contradiction. Otherwise, q_1 sent the DONE or REJECT message before receiving p's GRANT. Since q_1 only sends DONE or REJECT messages to processes from which it previously received a GRANT, then p sent another GRANT message to q_1 before t_G. This means that p sent two consecutive GRANT messages to q_1 without receiving a DONE or REJECT message in between. The only way this could happen is if p set its gid_p to -1 or p between sending the two GRANT messages without receiving a DONE or REJECT, which is a contradiction since this proof eliminated every other way of doing that.

Hence, p can not be $grants_{q_1}$ while p sends a GRANT message to q_2 at the same time, which conclude the proof of Lemma 4.1 . □

Lemma 4.2 (Self permission)**.** *Let p, q be two distinct alive processes. If $p \in grants_q$ then p can not enter CS.*

Proof. If $p \in grants_q$, then p sent a GRANT message to q and therefore set its gid_p to q. The reasoning of the proof for Lemma 4.1 can be used to show that p cannot change the value of its gid_p until q has removed p from its $grants_q$.

Since p is required to have its gid_p set to p in order to enter CS (line 8), then it is impossible for p to enter CS until after q removed p from $grants_q$. □

Claim 4.1 can now be proved by contradiction.

Proof. Let p_1, p_2 be two alive, distinct processes. Suppose that p_1 enters CS at time t_1, and p_2 enters CS at time t_2. Suppose that neither process leaves CS until after the other process has entered it. According to the live pairs intersection property of $\mathcal{T}\Sigma^{lr}$, there is a process q such that $q \in tq_{p_1}^{t_1} \cap tq_{p_2}^{t_2}$. It follows from the wait condition on line 8 that $q \in grants_{p_1}$ at time t_1 and $q \in grants_{p_2}$ at time t_2. There are two cases:

First case: p_1, p_2 and q are all distinct. Therefore, q sent a GRANT message to p_1 before t_1 and a GRANT message to p_2 before t_2. Additionally, neither process removed q from their *grants* set before entering CS. Without loss of generality, assume that q sent the GRANT message to p_1 first. There could be a run in which p_1 received the message immediately, and therefore added q to $grants_{p_1}$ before q sent the second GRANT to p_2. In this run, q sends a GRANT message to p_2 while $q \in grants_{p_1}$ at the same time, which is in contradiction with Lemma 4.1.

Second case: $q = p_1$ or $q = p_2$. Without loss of generality, assume that $q = p_1$. Since $q \in grants_{p_2}$ at time t_2, q sent a GRANT message to p_2 before t_2. Since it is impossible for q to send a GRANT message while in CS (because of the test on line 17), it follows that q sent the GRANT before entering CS. There could be a run in which p_2 received the GRANT immediately after it was sent, therefore adding q to $grants_{p_2}$ before q entered CS, which is in contradiction with Lemma 4.2. □

Claim 4.2 (Starvation freedom)**.** *If no process stays in its critical section forever, then every eventually up process that stopped crashing and reaches its try section will eventually enter its CS.*

To prove the Claim 4.2, the following lemmata is required:

Lemma 4.3 (Deadlock-free)**.** *Assuming that no process stays in CS forever, if a process p, which does not have the highest priority among the requesting processes, receives at least one* GRANT *from another process q, p will eventually either crash forever or remove q from $grants_p$, and q will eventually either crash forever or set gid_q to -1.*

Proof. Let p be a process in its try section at time t. There exists a distinct process p_h which is also in its try section at time t and has the highest priority among requesting processes.

Let q be a process distinct from p that sends a GRANT message that p receives at time t. It follows that p sent a REQUEST message to q at some time $t_R < t$.

One of the following cases applies:

1) p is eventually down, and q is not. Then according to the strong completeness property of $\mathcal{T}\Sigma^{lr}$, p will eventually be removed from tq_q and q will set gid_q to -1 on line 38.

2) q is eventually down, and p is not. Then according to the strong completeness property of $\mathcal{T}\Sigma^{lr}$, q will eventually be removed from tq_p and p will remove q from $grants_p$ on line 34.

3) At time t, $gid_p \neq -1$ and $gid_p \neq p$. Then when p receives q's GRANT message, it will never add q to $grants_p$ and will send q a REJECT message instead (line 46). When q receives the REJECT message, it will set gid_q to -1 (line 54).

4) At time t, $gid_p = -1$. When p calls CHECK REQUESTS, it will pass the test one line 17 since $requests_p$ contains at least p's request, and $\underline{crit}_p$ and $recovering_p$ cannot be *true* while in CS. p will then set gid_p to something different from -1 on line 18.

It follows from the cases above that the only way Lemma 4.3 could be false is if neither p nor q are eventually down, and $gid_p = p$ at time t. Since p is not eventually down, then p will eventually receive p_h's request at some time $t' > t$. Then one of the following cases applies:

1) During $[t_R, t']$, p does not crash, receives GRANT messages from every process in tq_p, and rdy_p is set to $\top$. Then p will end the wait on line 8 and enter CS. When p leaves CS, it will remove q from $grants_p$ on line 14 and send a DONE message to q on line 13. When q receives the DONE message, it will set gid_q to -1 on line 51.

2) During $[t_R, t']$, p does not crash and does not receive enough GRANT messages to enter CS (or rdy_p stays equal to $\perp$). Then at time t' when p receives p_h's request, it will call CHECK REQUESTS on line 43. p will pass the test on line 17 and, since p_h is the requesting process with the highest priority, p will set gid_p to p_h. It will then remove q from $grants_p$ on line 21 and send a REJECT message to q on line 22. When q receives the REJECT message, it will set gid_q to -1 on line 54.

3) During $[t_R, t']$, p crashes before receiving enough GRANT messages to enter CS. When p recovers, its $grants_p$ set is reinitialized and does not contain q. Since q was previously in tq_p and q is not eventually down, it follows from the trusting accuracy property of $\mathcal{T}\Sigma^{lr}$ that q is still in tq_p after p recovers. p will therefore send a COMEBACK message to q on line 27 with the $crit\_src$ parameter set to $false$. When q receives the COMEBACK message, it will set gid_q to -1 on line 59. Note that because of the first in, first out property, q will necessarily receive p's request before the COMEBACK message. Additionally, p will receive q's GRANT message before q's UPDATE message, and will ignore the grant because of the test on line 47. □

Lemma 4.4 (Decreasing priority)**.** *Assuming that no process stays in the CS forever, if an unstable process p is in the try section infinitely often, then the value of $round_p$ increases infinitely often (and therefore, p's priority decreases infinitely often).*

Proof. Let p be an unstable process that is in the try section infinitely often. By definition, p also crashes infinitely often. Let q be any eventually up process. According to the eventually strong accuracy property of $\mathcal{T}\Sigma^{lr}$, p will eventually trust q forever.

Let t_0 be a time after which every eventually down process crashed permanently, every eventually up process stopped crashing, and p started trusting q. According to the strong completeness property of $\mathcal{T}\Sigma^{lr}$, there is a time $t_1 \geq t_0$ such that $\forall t > t_1$, tq_p^t does not contain any eventually down process. Let $t_2 > t_1$ be the first time after t_1 that p crashes, and let $t_3 > t_2$ be the first time after t_2 that p enters the try section.

Every request sent by p after t_3 is sent only to processes that are not eventually down, including q. According to the dynamic connectivity property, q will receive every request sent by p after t_3. Every time that p crashes after t_3, p will send a COMEBACK message to q. Because of the first in, first out property, q will receive p's last request before receiving the COMEBACK message, and therefore when q receives the COMEBACK its $last\_round_q[p]$ will be up to date with q's latest $round_p$ value from before the crash. q will then respond with an UPDATE message, and p will update its $round_p$ value on line 63 before leaving the recovery phase. As a result, crashes after t_3 do not reduce or reset p's $round_p$ value.

At any time $t > t_3$, there are three possibilities:

1) p is in the exit or remainder section at time t. By assumption, p will eventually enter the try section, and therefore increase its $round_p$ value on line 4.

2) p is in the CS at time t. Since by assumption no process stays in the section forever, p will eventually leave CS and the case above applies.

3) p is in the try section at time t. Eventually, p will either enter CS (and the case above applies), or p will crash before entering the CS and therefore it will be in the remainder section after recovery (and the first case applies).

In all cases, there is a time $t' > t$ such that $round_p$ increases at time t'. □

Lemma 4.5 (Highest priority starvation freedom)**.** *Let t be a time after all eventually up processes stopped crashing. Assuming that no process stays in CS forever, if an eventually up process p is in the try section and has the highest priority among requesting eventually up processes at time t, then eventually p enters CS.*

Proof. Let p be an eventually up process that is in the try section with the highest priority among requesting eventually up processes at time t. By contradiction, assume that p never enters CS after t. It follows that p will never leave the try section, since it will neither crash nor enter CS. Therefore, p will never re-enter the try section and increase its $round_p$ value on line 4. It follows that p's priority will never change after t.

Let q_1 be any unstable process. According to Lemma 4.4, q_1 will either eventually stop entering the try section (in which case its priority becomes irrelevant), or q_1's priority will be reduced infinitely often, in which case p's priority will eventually be higher than q_1's. As a result, there is a time $t' \geq t$ after which p has the highest priority of all requesting processes in the system.

If $gid_p = q_2$ with q_2 distinct from q after t', then according to Lemma 4.3, eventually p will set its gid_p to -1 and then call CHECK REQUESTS. p will then set itself as gid_p on line 18 and will never change gid_p again.

According to the dynamic connectivity property, eventually every process in tq_p will have received p's request. Let q_3 be any process that received p's request. If $gid_{q_3} \neq -1$ and $gid_{q_3} \neq q_3$, then after t', according to Lemma 4.3, q_3 will eventually set gid_{q_3} to -1. When gid_{q_3} is equal to -1 or q_3 after t', then q_3 will set it to p on line 18 and send a GRANT message to p on line 19. As a result, p will receive a GRANT message from every process in tq_p.

Since p is eventually up, according to the quorum readiness property of $\mathcal{T}\Sigma^{lr}$, the eventually $rdy_p = \top$.

Finally, p will pass the wait condition on line 8 and enter CS, which is a contradiction. □

Claim 4.2 can now be proved.

Proof. Let p be an eventually up process that stopped crashing and is in its try section at time t. By contradiction, assume that p never enters CS after t. It follows that p will never leave the try section, since it will neither crash nor enter CS. Therefore, p will never re-enter the try section and increase its $round_p$ value on line 4. It follows that p's priority will never change after t, and that every requesting unstable process will eventually have a lower priority than p.

Let Q be the set of all requesting eventually up processes with higher priority than p. Let q be the process in Q with the highest priority. It follows from Lemma 4.5 that eventually, q will

enter CS. After q leaves CS, it will either (1) stop requesting forever (and therefore leave Q) or (2) enter the try section again and therefore decrease its priority. By induction, q will eventually not have the highest priority amongst requesting processes anymore, and another process in Q will take its place. As a result, eventually Q will become empty since every process in it will either stop requesting or increase its priority infinitely often.

Finally, p will become the requesting eventually up process with the highest priority, and according to Lemma 4.5, will enter CS, which is a contradiction. □

Claim 4.3 (Liveness)**.** *If an eventually up process p stopped crashing and is in the try section, then at some time later some process that is not eventually down is in CS.*

Proof. Let p be an eventually up process that stopped crashing and is in the try section. There are two possibilities:

- Some process eventually stays in CS forever. In this case, liveness is ensured.
- Otherwise, according to Claim 4.2, p will eventually enter CS, thus ensuring liveness.

□

Theorem 4.2 follows From Claim 4.1 and Claim 4.3:

Theorem 4.2 (Correctness)**.** *The Algorithm 5 solves the RME using $\mathcal{T}\Sigma^{lr}$ in any unknown dynamic environment.*

Corollary 4.1 (Sufficiency)**.** *The $\mathcal{T}\Sigma^{lr}$ failure detector is sufficient to solve the RME in any unknown dynamic environment with partial memory loss.*

4.4 Necessity of $\mathcal{T}\Sigma^{lr}$ to solve Fault-Tolerant Mutual Exclusion

This section proves that the $\mathcal{T}\Sigma^{lr}$ failure detector is necessary to solve the RME problem in any unknown dynamic system with partial memory loss. For this purpose, it can be assumed that there is an unknown dynamic system model $\mathcal{M}_{\text{RME}}$ with partial memory loss, in which RME can be solved with some algorithm $\mathcal{A}_{\text{RME}}$. The following proof will then show that the properties of $\mathcal{T}\Sigma^{lr}$ can be implemented in $\mathcal{M}_{\text{RME}}$.

Although the purpose of this section is to show that $\mathcal{T}\Sigma^{lr}$ can be implemented with RME in $\mathcal{M}_{\text{RME}}$, the same arguments and algorithms used here can also be used to show that $\mathcal{T}\Sigma^{l}$ can be implemented with fault-tolerant mutual exclusion in a static, known system without recovery. As a result, this section is also a part of the proof for Theorem 4.1.

The following proof is inspired by the proofs for the necessity of $\mathcal{T}$ and Σ^{l} in [DGFGK05] and [BCJ09], respectively. The main additional challenge is to merge the two proofs, since both trusting and quorum properties must apply for a same set tq_p.

The proof uses two algorithms, both of which share the following local variables:

$\underline{trust}_p \leftarrow \{p\}$ is the set of all processes that process p has heard of and that it does not suspect. This variable is in stable storage.

$start_p \leftarrow false$ is a flag used to delay the start of the RME algorithm.

Algorithm 6 Modified SEND primitive for $\mathcal{B}_{\text{RME}}$

```
1: procedure B_RME _SEND(msg, dest)
2:     wait for start_p = true
3:     SEND(msg, trust_p, dest)
4: upon reception of (msg, trust_src) from src do
5:     wait for start_p = true
6:     trust_p ← trust_p ∪ trust_src
7:     B_RME _DELIVER(msg)
```

First, algorithm $\mathcal{B}_{\text{RME}}$ is introduced. $\mathcal{B}_{\text{RME}}$ has exactly the same code as $\mathcal{A}_{\text{RME}}$, except that every call to the SEND primitive is replaced by a call to $\mathcal{B}_{\text{RME}}$ _SEND, as defined in Algorithm 6.

Algorithm 6 serves two purposes: (1) by using $\underline{trust}_p$, it enables p to keep track of which processes it heard of while trying to access CS; (2) by using $start_p$, it enables p to delay the start of the RME algorithm.

Lemma 4.6. *Provided that each eventually up process p eventually sets $start_p$ to true, Algorithm $\mathcal{B}_{\text{RME}}$ solves the RME problem in $\mathcal{M}_{\text{RME}}$.*

Proof. The only difference between $\mathcal{A}_{\text{RME}}$ and $\mathcal{B}_{\text{RME}}$ that could prevent $\mathcal{B}_{\text{RME}}$ from solving RME is the wait on lines 2 and 5. A process that never sets $start_p$ to *true* cannot participate in the algorithm. By assumption, this is only a problem for processes that are not eventually up. If a process never sets $start_p$ to *true*, then for the purpose of $\mathcal{B}_{\text{RME}}$, that process behaves exactly as an always down process would behave in a run of $\mathcal{A}_{\text{RME}}$. □

Algorithm 7 can now be introduced, which makes use of $\mathcal{A}_{\text{RME}}$ and $\mathcal{B}_{\text{RME}}$ to implement the properties of $\mathcal{T}\Sigma^{lr}$.

In addition to $\underline{trust}_p$ and $start_p$, Algorithm 7 uses following local variables:

$\underline{known}_p \leftarrow \{p\}$: as discussed in Section 4.1, $\underline{known}_p$ represents the knowledge that p has of other processes in the system. The algorithm does not show how $\underline{known}_p$ is kept up to date, but simply expects that $\underline{known}_p$ will eventually contain the process identities of (at least) all eventually up processes.

$\underline{crash}_p \leftarrow \emptyset$: the set of all processes that p is certain have crashed forever. Note that this variable is in stable storage.

$tq_p \leftarrow \emptyset$: the output of the $\mathcal{T}\Sigma^{lr}$ failure detector, which verifies the trusting and quorum properties.

$rdy_p \leftarrow \perp$: the other output variable of $\mathcal{T}\Sigma^{lr}$, which verifies the quorum properties.

$\underline{waitlist}_p \leftarrow \emptyset$: the set of processes to which p must grant permission for CS. This is used to ensure starvation freedom. Note that this variable is in stable storage.

$donelist_p \leftarrow \emptyset$: the set of processes to which p already granted permission for CS. It prevents p from always being passed over for CS access.

Algorithm 7 initially starts two tasks in parallel: TASK 1 and TASK 2. Later on, whenever process p gets knowledge of a process q, it starts a new task for q (denoted TASK 3 + q).

Each process p has its own CS, which is handled by algorithm $\mathcal{A}_{\text{RME}}$ and accessed with $\mathcal{A}_{\text{RME}}$.TRY(p). Additionally, there is a global CS which is handled by algorithm $\mathcal{B}_{\text{RME}}$ and accessed with $\mathcal{B}_{\text{RME}}$.TRY.

Algorithm 7 Reduction Algorithm $T_{\mathcal{A}_{\mathrm{RME}} \to \mathcal{T}\Sigma^{lr}}$: code for process p

1: **procedure** TASK 1
2: $\quad \mathcal{A}_{\mathrm{RME}}.\mathrm{TRY}(p)$
3: $\quad start_p \leftarrow true$
4: $\quad$ **loop forever:**
5: $\quad\quad$ **for** $q \in \underline{known}_p$ **do**
6: $\quad\quad\quad$ SEND(ALIVE, $req_p, \underline{trust}_p, q$)
7: **procedure** TASK 2
8: $\quad$ **loop forever:**
9: $\quad\quad$ **wait for** $\underline{waitlist}_p \setminus donelist_p = \emptyset$
10: $\quad\quad donelist_p \leftarrow \emptyset$
11: $\quad\quad req_p \leftarrow true$
12: $\quad\quad \mathcal{B}_{\mathrm{RME}}.\mathrm{TRY}$
13: $\quad\quad \mathcal{B}_{\mathrm{RME}}.\mathrm{EXIT}$
14: $\quad\quad req_p \leftarrow false$
15: $\quad\quad$ **if** $\underline{trust}_p \cap \underline{crash}_p = \emptyset$ **then**
16: $\quad\quad\quad tq_p \leftarrow \underline{trust}_p$
17: $\quad\quad\quad rdy_p \leftarrow \top$
18: $\quad\quad\quad$ **for** $q \in \underline{known}_p$ **do**
19: $\quad\quad\quad\quad$ SEND(QUORUM, $\underline{trust}_p, \underline{crash}_p, q$)
20: $\quad\quad$ **else**
21: $\quad\quad\quad \underline{trust}_p \leftarrow \underline{trust}_p \setminus \underline{crash}_p$
22: **procedure** TASK 3 + q
23: $\quad \underline{known}_p \leftarrow \underline{known}_p \cup \{q\}$
24: $\quad \mathcal{A}_{\mathrm{RME}}.\mathrm{TRY}(q)$
25: $\quad \mathcal{A}_{\mathrm{RME}}.\mathrm{EXIT}(q)$
26: $\quad \underline{crash}_p \leftarrow \underline{crash}_p \cup \{q\}$
27: **procedure** RECONNECTION
28: $\quad tq_p \leftarrow \underline{trust}_p \setminus \underline{crash}_p$
29: $\quad$ **for** $q \in \underline{trust}_p$ **do**
30: $\quad\quad$ **Start** TASK 3 + q
31: **when** $q \neq p$ is added to $\underline{trust}_p$
32: $\quad$ **Start** TASK 3 + q
33: **upon reception of** ALIVE $(req, trust\_src)$ **from** src **do**
34: $\quad \underline{trust}_p \leftarrow \underline{trust}_p \cup trust\_src$
35: $\quad$ **if** $req = true$ **then** $\underline{waitlist}_p \leftarrow \underline{waitlist}_p \cup \{src\}$
36: $\quad$ **else**
37: $\quad\quad \underline{waitlist}_p \leftarrow \underline{waitlist}_p \setminus \{src\}$
38: $\quad\quad donelist_p \leftarrow donelist_p \cup \{src\}$
39: **upon reception of** QUORUM $(trust\_src, crash\_src)$ **from** src **do**
40: $\quad \underline{trust}_p \leftarrow \underline{trust}_p \cup trust\_src$
41: $\quad \underline{crash}_p \leftarrow \underline{crash}_p \cup crash\_src$
42: $\quad$ **if** $rdy_p = \bot$ **then**
43: $\quad\quad tq_p \leftarrow \underline{trust}_p \setminus \underline{crash}_p$

In TASK 1, p enters its own CS and then never leaves it. Since in this case a well-formed process restarts in the CS after a recovery, this means that a recovering process will restart TASK 1 directly after line 2 if it previously managed to enter its own CS. This enables other processes to detect p's failure if it crashes permanently (if another process manages to access p's CS in TASK $3 + p$, it means that p crashed forever). In TASK 1, p also sends information to the rest of the system about its own identity and whether or not p is trying to access the global CS. These ALIVE messages are used by other processes to keep $\underline{trust}_p$, $\underline{waitlist}_p$, and $donelist_p$ up to date.

In TASK 2, p tries infinitely often to access the global CS. The wait on line 9 helps to ensure that the global CS starvation freedom property is satisfied. After entering and leaving the global CS, if p entered it using only messages from processes that are not crashed (test on line 15), then p updates its $\mathcal{T}\Sigma^{lr}$ output variables and informs other processes with QUORUM messages. However, if p used information from crashed processes to enter CS, it removes them from its $\underline{trust}_p$ set.

TASK $3 + q$ is started by p when q is added to $\underline{trust}_p$, and is used to detect q's permanent crash.

When a process p receives a QUORUM message, it updates its local $\underline{trust}_p$ and $\underline{crash}_p$ information and, if rdy_p is currently $\perp$ (and therefore p is not currently trying to verify the live pairs intersection property), then p updates its tq_p.

Lemma 4.7 (Starvation freedom)**.** *Every eventually up process passes the lines 12 – 13 infinitely often.*

Proof. By contradiction, assume that there is an eventually up process p which does not go through CS infinitely often. There are two ways this could happen: p is either stuck in the wait on line 9 forever, or p is stuck in try section on line 12 forever.

First assume that p is stuck in try section forever. Since the liveness property of RME is verified, and since no process can stay in CS forever (since the CS has no code), it follows that there is a process q that enters CS infinitely often.

Eventually, $p \in \underline{known}_q$ and $q \in \underline{known}_p$. Since p set req_p to $true$ on line 11, then eventually q will receive an ALIVE message from p with req set to $true$, and q will add p to $\underline{waitlist}_q$. Because of the first in, first out property, q will eventually stop receiving any ALIVE message from p that has the req value set to $false$. Since q passes the line 10 infinitely often, eventually $p \notin donelist$. Since $p \in \underline{waitlist}_q \setminus donelist_q$, then eventually q will wait forever on line 9, which is a contradiction.

Now assume that p is stuck on line 9 forever. Let W be the set of processes that stay in $\underline{waitlist}_p \setminus donelist_p$ for infinitely long. Note that a process q that is not stuck forever in the try section on line 12 would have their req set to $false$ and therefore would send an ALIVE message to p with req set to $false$, and would be removed from $\underline{waitlist}_p \setminus donelist_p$ as a result. It follows that every $q \in W$ is stuck forever on line 12. If q is eventually down, it eventually crashes forever and therefore cannot be in W. If q is eventually up, according to the previous paragraph it eventually enters CS and therefore cannot be in W. If q is unstable, it eventually crashes and resets its req_q to $false$, and therefore cannot be in W. As a result, W is empty and p eventually ends the wait on line 9.

□

Lemma 4.8 (Crashed completeness)**.** *A process can only be added to $\underline{crash}_p$ if it crashed forever.*

Proof. A process can only be added to $\underline{crash}_p$ on lines 26 and 41. In order for p to add a process to $\underline{crash}_p$ on line 41, some other process q must have added it to $\underline{crash}_q$ on line 26 first.

In order for p to add a process q to $\underline{crash}_p$ on line 26, p must first have started TASK $3 + q$. This can only happen if p added q to $\underline{trust}_p$. A process can be added to $\underline{trust}_p$ on lines 34 and 40, or by receiving information from q as part of algorithm $\mathcal{B}_{\text{RME}}$. If q sent a QUORUM message, then it must have passed the CS on lines 12 – 13 and therefore sent or received information as part of algorithm $\mathcal{B}_{\text{RME}}$, which means that $start_q$ was set to $true$. Whether q set $start_q$ to $true$ on line 3 or sent an ALIVE message on line 6, it had to enter its own CS on line 2 first.

Since q entered its own CS before p started TASK $3 + q$ and will never leave it, the only way that p can reach line 26 and add q to $\underline{crash}_p$ is if q crashed forever. □

Claim 4.4 (Strong completeness)**.** *Algorithm 7 ensures the strong completeness property of $\mathcal{T}\Sigma^{lr}$ in $\mathcal{M}_{\text{RME}}$.*

Proof. Let p be an eventually down process, and q be a process that is not eventually down. Note that by construction, a process can never be added to tq_q without being added to $\underline{trust}_q$ first. There are two cases:

p was never added to $\underline{trust}_q$. Then the property is immediately verified.

p was added to $\underline{trust}_q$. Let r be some eventually up process. Eventually, q will send an ALIVE message to r which contains $\underline{trust}_q$. Therefore, r will eventually add p to its $\underline{trust}_r$ and will then start TASK $3 + p$. After p crashes forever, eventually r will reach line 26 and add p to $\underline{crash}_r$.

Let t_1 be a time after which all eventually down processes have crashed. Let $t_2 \geq t_1$ be a time after which there are no more messages sent by eventually down processes in the system. After t_2, neither q nor r will ever add an eventually down process into their $\underline{trust}$ set again. According to Lemma 4.7, r will then eventually remove all eventually down processes from $\underline{trust}_r$ on line 21. Since according to Lemma 4.8 only eventually down processes can be in $\underline{crash}_r$, after this time r will always pass the test on line 15 and therefore r will send a QUORUM message to q infinitely often.

If q goes through the loop in TASK 1 infinitely often, it will act like r and eventually never have p in its tq_q. If q is unstable and does not go through the loop in TASK 1 infinitely often, then after it stops going through the loop it will crash and reset its rdy_q to $\perp$. Then, the next time that q receives a QUORUM message from r, it will add p to $\underline{crash}_q$ and remove it from tq_q on line 43 □

Claim 4.5 (Eventually strong accuracy)**.** *Algorithm 7 ensures the eventually strong accuracy property of $\mathcal{T}\Sigma^{lr}$ in $\mathcal{M}_{\text{RME}}$.*

Proof. Let p be an eventually up process, and q a process that is not eventually down. Eventually, $q \in \underline{known}_p$. According to the liveness property of RME, p will eventually enter its own CS and send an ALIVE message to q on line 6. When q receives the message, it will add p to its $\underline{trust}_q$ set on line 34. It follows from Lemma 4.8 that p will never be in $\underline{crash}_q$. According to the proof for Claim 4.4, q will update its tq_p infinitely often with $\underline{trust}_q$, either on line 16 or on line 43. As a result, $p \in tq_q$ forever. □

Claim 4.6 (Trusting accuracy)**.** *By construction, the only way that a process can be removed from tq_p is by being added to $\underline{crash}_p$. The proof then follows directly from Lemma 4.8.*

Claim 4.7 (Quorum readiness)**.** *Algorithm 7 ensures the quorum readiness property of $\mathcal{T}\Sigma^{lr}$ in $\mathcal{M}_{\text{RME}}$.*

Proof. Let p be an eventually up process. According to the proof for Claim 4.4, p will pass the test on line 15 infinitely often. After p stops crashing, the next time it reaches line 17, it will set rdy_p to $\top$ forever. □

Lemma 4.9 (Message reception intersection)**.** *Let p_1 and p_2 be two processes that enter the CS of $\mathcal{B}_{\text{RME}}$ at time t_1 (resp. t_2). Let Q_1 (resp. Q_2) be the set of all processes from which p_1 (resp. p_2) received information from (directly or through forwarding) since the last time it entered the try section before t_1 (resp. t_2). Then either one of the process crashed permanently before the other entered CS, or $Q_1 \cap Q_2 \neq \emptyset$.*

Proof. By contradiction, assume that $Q_1 \cap Q_2 = \emptyset$.

First assume that in $\mathcal{B}_{\text{RME}}$, a process r might send a message to a process s to authorize s to enter CS before s has entered the try section. In this case, it is possible that every process in the system would send such a message to s before s enters the try section. Now consider a run in which a process s' different from s later enters the try section. If $\mathcal{B}_{\text{RME}}$ allows some process to authorize s', then all other processes might do the same thing. As a result, if s is not permanently crashed, s and s' might enter CS at the same time, thus violating the safety property. If $\mathcal{B}_{\text{RME}}$ does not allow any process to authorize s', then s might never enter the try section, thus violating the liveness property. It follows that in $\mathcal{B}_{\text{RME}}$, only messages received after entering the try section can authorize a process to enter CS.

Now consider a run in which every message between Q_1 and Q_2 is delayed until after both p_1 and p_2 have left CS. This means that the system is partitioned, and therefore algorithm $\mathcal{B}_{\text{RME}}$ cannot possibly prevent a run in which both p_1 and p_2 enter CS at the same time, thus violating the safety property of RME. □

Claim 4.8 (Live pairs intersection)**.** *Algorithm 7 ensures the live pairs intersection property of $\mathcal{T}\Sigma^{lr}$ in $\mathcal{M}_{\text{RME}}$.*

Proof. The live pairs intersection property only applies when rdy_p is set to $\top$, and the only way to set rdy_p to $\top$ is on line 17. Since lines 28 and 43 can only be reached when rdy_p is set to $\bot$, it follows that at any time rdy_p is equal to $\top$, the current value of tq_p was set on line 16.

Note that tq_p is set from $\underline{trust}_p$ on line 16 after p recently went through the global try, critical, and exit sections with $\mathcal{B}_{\text{RME}}$ on lines 12 – 13. By construction, every process from which p received information (even indirectly) in $\mathcal{B}_{\text{RME}}$ since last entering the try section is in $\underline{trust}_p$ at that time. Observe also that the only way to remove a process identity from $\underline{trust}_p$ is on line 21, which cannot be reached between lines 12 and 16.

Let p_1 and p_2 be two processes, and let t be some time at which both are alive. Then for any time $t_1 \leq t$ when p_2 reached line 16 and any time $t_2 \leq t$ when p_2 reached line 16, it follows from Lemma 4.9 that $\underline{trust}_{p_1}$ at time t_1 and $\underline{trust}_{p_2}$ at time t_2 intersect. □

From Claims 4.4 to 4.8, the following theorem can be deduced:

Theorem 4.3 (Correctness)**.** *The Algorithm 7 implements $\mathcal{T}\Sigma^{lr}$ in $\mathcal{M}_{\mathrm{RME}}$.*

Corollary 4.2 (Necessity)**.** *The $\mathcal{T}\Sigma^{lr}$ failure detector is necessary to solve the RME in any unknown dynamic environment with partial memory loss.*

Conclusion

This chapter adapted the $(\mathcal{T}, \Sigma^l)$ failure detector into the $\mathcal{T}\Sigma^{lr}$ failure detector adapted to unknown dynamic systems with partial memory loss and where faulty processes may recover. $\mathcal{T}\Sigma^{lr}$ was proved to be both necessary and sufficient to solve the RME problem in such systems and it is, therefore, the weakest failure detector to solve RME in unknown dynamic systems with partial memory loss.

Additionally, this chapter showed that the properties of $(\mathcal{T}, \Sigma^l)$ can apply to two separate output variables or to a single one, without changing the strength of the failure detector.

Chapter 5

Conclusion

Contents

5.1 Contributions 78

5.1.1 A Failure Detector for k-Set Agreement in Unknown Dynamic Systems 78

5.1.2 A Failure Detector for Recoverable Mutual Exclusion in Unknown Dynamic Systems 78

5.2 Perspectives 79

5.2.1 On the Necessity of Synchronous Processes in Dynamic Systems 79

5.2.2 The Weakest Failure Detector for k-Set Agreement 79

5.2.3 Defining the Mutual Exclusion Problem in Crash-Recovery Systems . . 79

Unknown dynamic systems pose new challenges to the field of distributed computing, and failure detectors in particular. The lack of initial information in an unknown system, along with the dynamics of the set of processes and the communication graph, prevent existing failure detector results from being applied to these new systems. For instance, some detectors, such as Σ and Σ^l, can not be implemented without initial information on the participants in the system. Additionally, in order to be implemented, all failure detectors require a specific level of temporal graph connectivity. Furthermore, even in a system enriched with a failure detector, no problem can be solved without the assumption of some level of temporal graph connectivity that is specific to the problem. In other words, if the processes in the upper layer application can not communicate with each other, then no problem can be solved even with perfect information on failures.

This thesis addressed these challenges by providing new definitions of previously existing failure detectors that are suitable to unknown dynamic systems, and used them to solve two fundamental problems of distributed computing: k-set agreement and RME. To circumvent the lack of initial information, the $\perp$ default value was added as a way to delay the failure detector outputs. In Chapter 3, a failure detector implementation was proposed which makes use of the TVG model of [CFQS12] to express the required level of temporal graph connectivity. Finally, the necessity of temporal graph connectivity assumptions to solve problems was addressed differently in Chapters 3 and 4.

In Chapter 3, the connectivity assumptions used to solve k-set agreement were integrated into the definition of the failure detector itself. This approach presents the advantage of preserving the role of the failure detector as a complete abstraction layer: no other assumption than the failure detector is used to solve the problem. Conversely, in Chapter 4, the connectivity assumptions used to solve RME were presented as part of the base system model. While this approach limits the applicability of the results to systems making the same connectivity assumptions, it presents the advantage of making the new failure detector definition closer to its existing definition for static and known systems. This allowed Chapter 4 to be a direct translation of the existing results of [DGFGK05] and [BCJ09] for unknown dynamic systems.

5.1 Contributions

This section summarizes the contributions presented in this thesis.

5.1.1 A Failure Detector for k-Set Agreement in Unknown Dynamic Systems

Chapter 3 adapted the definition of the $\Pi\Sigma_{x,y}$ failure detector into the $\Pi\Sigma_{\perp,x,y}$ failure detector. This new detector differs from the original in two ways: it makes use of the $\perp$ default value to deal with the lack of initial information about the participants in the system, and it includes two connectivity properties (one is the quorum connectivity property, and the other is integrated into the eventual partial leadership property).

Using the TVG formalism enriched with message pattern assumptions to model an unknown dynamic system, an implementation of $\Pi\Sigma_{\perp,x,y}$ was then presented.

Additionally, an algorithm adapted from the k-set agreement algorithm of [BT10, MRS12] was proposed. This algorithm solves the k-set agreement problem in any unknown dynamic system enriched with $\Pi\Sigma_{\perp,x,y}$ with $k \leq xy$, thus proving that this failure detector is sufficient to solve k-set agreement in unknown dynamic systems.

5.1.2 A Failure Detector for Recoverable Mutual Exclusion in Unknown Dynamic Systems

In Chapter 4, graph connectivity issues were abstracted into model assumptions. As a result, this work focused on process mobility, which was modeled using the crash-recovery failure model.

The chapter first discussed the differences between $(\mathcal{T}, \Sigma^l)$ and $\mathcal{T}\Sigma^l$ (which requires the properties of $\mathcal{T}$ and Σ^l to apply to a same output set), and proved that both failure detectors are equivalent.

Based on that result, $\mathcal{T}\Sigma^l$ was then extended into the $\mathcal{T}\Sigma^{lr}$ failure detector, which modifies the former in two ways: it uses of the $\perp$ default value (in the form of the rdy_p variable) and is defined in terms of eventually up, eventually down and unstable processes, instead of correct and faulty processes.

An algorithm that solves RME using $\mathcal{T}\Sigma^{lr}$ was proposed, thus proving that it is sufficient to solve RME in unknown dynamic systems. Finally, a reduction algorithm transforming any failure detector sufficient to solve RME into $\mathcal{T}\Sigma^{lr}$ was also presented, thus proving that $\mathcal{T}\Sigma^{lr}$ is the weakest failure detector to solve RME in unknown dynamic systems.

5.2 Perspectives

5.2.1 On the Necessity of Synchronous Processes in Dynamic Systems

Chapter 3 made the assumption that processes are synchronous. This assumption ensures that processes send their messages often enough to take advantage of the communication link availability windows. If this synchrony assumption is removed, even if a journey between two processes is available infinitely often, there is no guarantee that processes will ever manage to take advantage of this journey to get messages through.

Some papers in the literature have made a different assumption for the same purpose, like in [GCLL15], where it is assumed that new edges appearing in the graph are detected fast enough by processes to make use of the edge availability window. Such an assumption also implies some common timing between processes, since the definition of "fast enough" must be the same system-wide.

It would seems, therefore, that some sort of process synchrony assumption is necessary to reliably exploit communication opportunities in dynamic systems, but no such result has been proved so far.

5.2.2 The Weakest Failure Detector for k-Set Agreement

There is no proof that $\Pi\Sigma_{x,y}$ (in static and known systems) or $\Pi\Sigma_{\perp,x,y}$ (in unknown dynamic systems) is the weakest failure detector for k-set agreement in the general case of $1 \leq k < n$.

Although the weakest failure detectors for consensus ($k = 1$) and set agreement ($k = n - 1$) have been identified, the weakest failure detector for $2 \leq k < n - 1$ has yet to be found. This issue has been the object of many of the papers presented in Section 2.3, and is still an open problem.

5.2.3 Defining the Mutual Exclusion Problem in Crash-Recovery Systems

Chapter 4 focused on a definition of RME that allows temporarily down processes to stay in the critical section until they recover and finish it. On the other hand, a process that crashes permanently must release the critical section in order to ensure liveness. This version of the problem is difficult to solve, since it implies that processes must be able to detect whether the crash of another process is temporary or permanent.

This version of RME is powerful because it ensures that process can always execute their critical section without being interrupted by other processes which is important, for example, to preserve the consistency of a shared resource. However, some applications might not require this level of consistency.

Another, weaker version of RME could be defined in which a process that crashes always leaves the critical section, whether the crash is temporary or permanent. This version of the problem is easier to solve, and requires a weaker version of the trusting accuracy property, since distinguishing a permanent crash from a temporary one would no longer be necessary.

Further research could aim to define the weakest failure detector for this weaker mutual exclusion problem.

Appendix A

A Reliable Broadcast Protocol for Asynchronous Systems with a Hypercube Topology

The following is an article [JRAJ16] that was written with Luiz A. Rodrigues, Luciana Arantes and Elias Procópio Duarte Jr. as part of the joint project CNRS - Fundação Araucária between the LIP6 (Delys team), the Universidade Federal do Paraná (UFPR) and the Universidade Estadual do Oeste do Paraná (UNIOESTE) in Brazil. Although it is a distributed computing paper, it does not concern dynamic systems and it is therefore presented as an appendix to this thesis.

Contents

A.1 Introduction 81
A.2 Related Work 83
A.3 System Model 85
A.4 The VCube 85
A.5 Reliable Broadcast Algorithm for Asynchronous System 86
A.5.1 Message types and local variables 87
A.5.2 Algorithm description 88
A.5.3 Proof of correctness 90
A.6 Performance Discussion 92
A.7 Conclusion and Future Work 93

A.1 Introduction

Numerous distributed applications with information dissemination requirements rely on a broadcast communication primitive to send messages to all processes that compose the application

[BDPB13]. Formally, reliable broadcast is defined in terms of two primitives: *broadcast(m)*, which is defined by the broadcast algorithm and called by the application to disseminate m to all processes, and *deliver(m)*, which is defined by the application and called by the broadcast algorithm when message m has been received. The broadcast algorithm that offers these primitives must ensure that, if a correct[1] process broadcasts a message, then it eventually delivers the message (*validity* property). Furthermore, every correct process delivers a message at most once and only if that message was previously broadcast by some process (*integrity* property).

From an implementation point of view, the broadcast primitive sends point-to-point messages to each process of the system. However, if the sender fails during the execution of the broadcast primitive, some processes might not receive the broadcast message. In order to circumvent this problem, *reliable broadcast* ensures, besides the *validity* and *integrity* properties, that even if the sender fails, every correct process delivers the same set of messages (*agreement* property) [HT93].

There exists a considerable amount of literature on reliable broadcast algorithms, such as the one where all correct receivers retransmit all received messages guaranteeing then the delivery of all broadcast messages by the other correct processes of the system [GR06]. We are particularly interested in solutions that use failure detectors[CT96] which notify the broadcast algorithm about processes failures. Upon receiving such an information, the algorithm reacts in accordance to tolerate the failure. Another important feature of reliable broadcast algorithms concerns performance, which is related to how broadcast messages are diffused to processes. Aiming at scalability and message complexity efficiency, many reliable broadcasts organize processes on logical spanning trees. Messages are then diffused over the constructed tree, therefore providing logarithmic performance [SGS84], [FA96], [KMV10], [RS88], [RSCW14] (see Section A.2).

This work presents an autonomic reliable broadcast algorithm where messages are transmitted over spanning trees dynamically built on top of a logical hierarchical hypercube-like topology. Autonomic systems constantly monitor themselves and automatically adapt to changes [KC03]. The logical topology is maintained by the underlying VCube monitoring system which also detects failures [DBR14] [DJN98]. VCube is a distributed diagnosis layer responsible for organizing processes of the system in a virtual hypercube-like cluster-based topology which is dynamically re-organized in case of process failure. When invoked, the VCube gives information about the liveness of the processes that compose the system.

We assume a fully-connected **asynchronous** system in which processes can fail by crashing, and crashes are permanent. Links are reliable. A process that invokes the reliable broadcast primitive starts the construction of a spanning tree. This tree is built with information obtained from the VCube, and is dynamically reconstructed upon detection of a node crash (process failure).

In a previous work [RAJ14], we proposed an autonomic reliable broadcast algorithm on top of the Hi-ADSD, a previous version of the VCube. The algorithm guarantees several logarithmic properties, even when nodes fail, and allows transparent and efficient spanning tree reconstructions. However, for this solution, we considered a synchronous model for the system, i.e., there exist known bounds on message transmission delays and processors' speed and, consequently, the VCube needs to provide perfect process failure detections. On the one hand, the advantage of such synchronous assumption is that there was no false failure suspicions and, thus, if the

[1] A correct process is a process that does not fail during execution

VCube notifies the broadcast algorithm that a given process is faulty, the algorithm is sure that it can stop sending message to this faulty process and then removes it forever from the spanning tree constructions. On the other hand, the synchronous assumption considerably restrains the distributed systems and applications that can use the broadcast protocol since many of the current network environments are considered asynchronous (there exist no bounds on message transmission delay or on processors' speed).

Hence, considering the above constraints, we propose in this article a new autonomic reliable broadcast algorithm, using the VCube in an asynchronous model. We assume that the failure detection service provided by the VCube is unreliable since it can make mistakes by erroneously suspecting a correct process (false suspicion) or by not suspecting a node that has actually crashed. However, upon detection of its mistake, the VCube corrects it. Furthermore, it also ensures that eventually all failures are detected (*strong completeness* property). Note that such false suspicions render a broadcast algorithm much more complex than the previous one since it can induce violation of the properties. For instance, the algorithm must ensure that a falsely suspected process must receive and deliver, only once, all broadcast messages, otherwise the *agreement* and *integrity* properties would be violated. In our solution, false suspicions are tolerated by sending special messages to those processes suspected of having failed. We must also emphasize that our aim is to provide a reliable broadcast algorithm which is efficient, i.e., that keeps, as much as possible, the logarithmic properties of the spanning tree diffusion over the hypercube-like topology. Our algorithm tolerates up to n-1 node crashes.

The rest of this paper is organized as follows. Section A.2 discusses some related work. In Section A.3 we describe the system model while Section A.4 briefly describes the VCube diagnosis algorithm and the hypercube-like topology. In Section A.5, we present the autonomic reliable broadcast algorithm for asynchronous systems while Section A.6 discuss some performance issue of the algorithm. Finally, Section A.7 concludes the paper.

A.2 Related Work

Many reliable broadcast algorithms of the literature exploit spanning trees such as [SGS84, FA96, KMV10, RS88, RSCW14].

Schneider et. al. introduced in[SGS84] a tree-based fault-tolerant broadcast algorithm whose root is the process that starts the broadcast. Each node forwards the message to all its successors in the tree. If one process p that belongs to the tree fails, another process assumes the responsibility of retransmitting the messages that p should have transmitted if it were correct. Like to our approach, processes can fail by crashing and the crash of any process is detected after a finite but unbounded time interval by a failure detection module. However, the authors do not explain how the algorithm rebuilds or reorganizes the tree after a process failure.

In [FA96], a reliable broadcast algorithm is provided by exploiting disjoint paths between pairs of source and destination nodes. Multiple-path algorithms are particularly useful in systems that cannot tolerate the time overhead for detecting faulty processors, but there is an overhead in the number of duplicated messages. On a star network with n edges, the algorithm constructs $n-1$ directed edge-disjoint spanning trees. Fault tolerance is achieved by retransmitting the same messages through a number of edge-disjoint spanning trees. The algorithm tolerates up to $n-2$

failure of nodes or edges and can be adjusted depending on the network reliability. Similarly, Kim et al. propose in [KMV10] a tree-based solution to disseminate a message to a large number of receivers using multiple data paths in a context of time-constrained dissemination of information. Thus, arguing that reliable extensions using ack-based failure recovery protocols cannot support reliable dissemination with time constraints, the authors exploit the use of multiple data paths trees in order to conceive a fast and reliable multicast dissemination protocol. Basically the latter is a forest-based (multiple parents-to-multiple children) tree structure where each participant node has multiple parents as well as multiple children. A third work that exploits multi-paths spanning trees is [RS88] where the authors present a reliable broadcast algorithm that runs on a hypercube and uses disjoint spanning trees for sending a message through multiple paths.

Raynal et. al. proposed in [RSCW14] a reliable tree-based broadcast algorithm suited to dynamic networks in which message transfer delays are bounded by a constant of δ unit of times. Whenever a link appears, its lifetime is at least δ units of time. The broadcast is based on a spanning-tree on top of which processes forward received messages to their respective neighbors. However, as the system is dynamic, the set of current neighbors of a process p may consists of a subset of all its neighbors and, therefore, p has to additionally execute specific statements when a link re-appears, i.e., forwards the message on this link if it is not sure that the destination process already has a copy of it.

Similarly to our approach, many existing reliable broadcast algorithms exploit spanning trees constructed on hypercube-like topologies [RS88, Wu96, LB99]. In [Wu96], the authors present a fault-tolerant broadcast algorithm for hypercubes based on binomial trees. The algorithm can recursively regenerate a faulty subtree, induced by a faulty node, through one of the leaves of the tree. On the other hand, unlike our approach, there is a special message for advertising that the tree must be reconstructed and, in this case, broadcast messages are not treated by the nodes until the tree is rebuilt. The HyperCast protocol proposed by [LB99] organizes the members of a multicast group in a logical tree embedded in a hypercube. Labels are assigned to nodes and the one with the highest label is considered to be the root of the tree. However, due to process failures, multiple nodes may consider themselves to be the root and/or different nodes may have different views of which node is the root.

Leitão et al. present in [LPR07] the *HyParView*, a hybrid broadcast solution that combines a tree-based strategy with a gossip protocol. A broadcast tree is created embedded on a gossip-based overlay. Broadcast is performed by using gossip on the tree branches. Later, some of the authors proposed a second work [FLR10] where they introduced *Thicket*, a decentralized algorithm to build and maintain multiple trees over a single unstructured P2P unstructured overlay for information diffusion. The authors argue that multiple trees approach allow that each node to be an internal node in just a few trees and a leaf node in the remaining of the trees providing, thus, load distribution as well as redundant information for fault-tolerance.

In [RAJ14], we presented a reliable broadcast solution based on dynamic spanning trees on top of the Hi-ADSD, a previous version of the VCube. Multiple trees are dynamically built, including all correct nodes, where each tree root corresponds to the node that called a broadcast primitive. Contrarily to the current work, this solution considers that the system model is synchronous and that the VCube offers a perfect failure detection.

A.3 System Model

We consider a distributed system that consists of a finite set P of $n > 1$ processes. Each process has a unique address. Processes $\{p_0, .., p_{n-1}\}$ communicate only by message passing. Each single process executes one task and runs on a single processor. Therefore, the terms node and process are used interchangeably in this work.

The system is asynchronous, i.e., relative processor speeds and message transmission delay are unbounded. Links are reliable, and, thus, messages exchanged between any two correct processes are never lost, corrupted or duplicated. There is no network partitioning.

Processes communicate by sending and receiving messages. The network is fully connected: each pair of processes is connected by a bidirectional point-to-point channel. Processes are organized in a virtual hypercube-like topology, called VCube. In a d-dimensional hypercube (d-cube) each process is identified by a binary address i_{d-1}, i_{d-2}, i_0. Two processes are connected if their addresses differ by only one bit. Processes can fail by crashing and, once a process crashes, it does not recover. If a process never crashes during the run, it is considered *correct* or *fault-free*; otherwise it is considered to be *faulty*. After any crash, the topology changes, but the logarithmic properties of the hypercube are kept.

We consider that the primitives to send and receive a message are atomic, but the broadcast primitives are not.

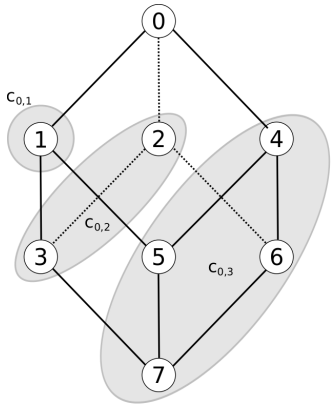
A.4 The VCube

Let n be the number of processes in the system P. VCube [DBR14] is a distributed diagnosis algorithm that organizes the correct processes of the system P in a virtual hypercube-like topology. In a hypercube of d dimensions, called d-VCube, there are 2^d processes. A process i groups the other $n-1$ processes in $\log_2 n$ clusters, such that cluster number s has size 2^{s-1}. The ordered set of processes in each cluster s is denoted by $c_{i,s}$ as follows, in which $\oplus$ denotes the bitwise exclusive *or* operator (xor).

$$c_{i,s} = \{i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1}\} \tag{A.1}$$

A process i tests another process in the $c_{i,s}$ to check whether it is correct or faulty. It executes a test procedure and waits for a reply. If the correct reply is received within an expected time interval, the monitored process is considered to be alive. Otherwise, it is considered to be faulty. We should point out that in an asynchronous model, which is the case in the current work, VCube provides an ureliable failure detection since it can erroneously suspect a correct process (false suspicion). If later it detects its mistake, it corrects it. On the other hand, according to the properties proposed by Chandra and Toueg [CT96] for unreliable failure detectors, the VCube ensures the *strong completeness* property: eventually every process that crashes is permanently suspected by every correct process. Since there are false suspicions, the VCube does not provide any *accuracy* property. A VCube providing both *completeness* and *accuracy* could not possibly be implemented in a fully asynchronous system, according to Fischer, Lynch and Paterson [FLP85].

Timestamps are used to identify the latest state of the tested processes. Based on the replies of the tests, process i connects itself to one fault-free process of each cluster s, if it exists. If



s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figure A.1: VCube hierarchical organization.

there are no failures, a complete logical hypercube is created.

Fig. A.1 shows the hierarchical cluster-based logical organization of $n = 8$ processes connected by a 3-VCube topology as well as a table which contains the composition of all $c_{i,s}$ of the 3-VCube.

Let's consider process p_0 and that there are no failures. The clusters of p_0 are shown in the same figure. Each cluster $c_{0,1}$, $c_{0,2}$, and $c_{0,3}$ is tested once, i.e., p_0 only performs tests on nodes $1, 2, 4$ which will then inform p_0 about the state of the other nodes of the respective cluster.

In order to avoid that several processes test the same processes in a given cluster, process i executes a test on process $j \in c_{i,s}$ only if process i is the first faulty-free process in $c_{j,s}$. Thus, any process (faulty or fault-free) is tested at most once per round, and the latency, i.e., the number of rounds required for all fault-free processes to identify that a process has become faulty is $\log_2 n$ in average and $\log_2^2 n$ rounds in the worst case.

A.5 Reliable Broadcast Algorithm for Asynchronous System

A reliable broadcast algorithm ensures that the same set of messages is delivered by all correct processes, even if the sender fails during the transmission. Reliable broadcast presents three properties [GR06]:

- *Validity*: if a correct process broadcasts a message m, then it eventually delivers m.
- *Integrity*: every correct process delivers the same message at most once (no duplication) and only if that message was previously broadcast by some process (no creation).
- *Agreement*: if a message m is delivered by some correct process p_i, then m is eventually delivered by every correct process p_j. Note that the agreement property still holds if m is not delivered by any process.

Our reliable broadcast algorithm exploits the virtual topology maintained by VCube, whenever possible. Each process creates, thus, a spanning tree rooted at itself to broadcast a message. The message is forwarded over the tree and, for every message that a node of the tree sends to one of its correct neighbor, it waits for the corresponding acknowledge from this neighbor, confirming the reception of the message. Algorithm 8 presents the pseudo-code of our proposal reliable broadcast protocol for an asynchronous system with n=2^d processes. The dimension of the VCube is, therefore, d. A process gets information, not always reliable, about the liveness of the other processes by invoking the VCube. Hence, the trees are dynamically built and autonomically maintained using the hierarchical cluster structure and the knowledge about faulty (or falsely faulty suspected) nodes. The algorithm tolerates up to n-1 failures.

Let i and j be two different processes of the system. The function $cluster_i(j) = s$ returns the identifier s of the cluster of process i that contains process j, $1 \leq s \leq d$. For instance, in the 3-cube as shown in Fig. A.1, $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$ and $cluster_0(4) = cluster_0(5) = cluster_0(6) = cluster_0(7) = 3$.

A.5.1 Message types and local variables

Let m be the application message to be transmitted from a sender process, denoted *source*, to all other processes in the system. We consider three types of messages:

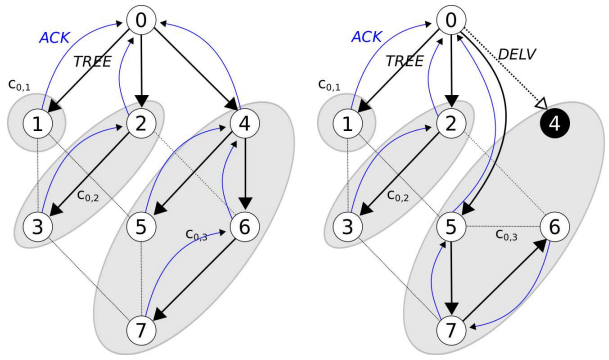
- $\langle TREE, m\rangle$: message broadcast by the application that should be forwarded over the VCube to all processes considered to be correct by the sender;
- $\langle DELV, m\rangle$: message sent to processes suspected of being faulty in order to avoid that false suspicions induce the no delivery of the message by correct processes. The recipient of the message should deliver it but not forward it;
- $\langle ACK, m\rangle$: used as an acknowledgement to confirm that a TREE message related to m was received.

For the sake of simplicity, we use TREE, DELV, and ACK to denote these messages.

Every message m keeps two parameters: (1) the identifier of the process that broadcast m and (2) a *timestamp* generated by the process local counter which uniquely identifies the m. The first message broadcast by a process i has timestamp 0 and at every new broadcast, i increments the timestamp by 1. The algorithm can extract these two parameters from m by respectively calling the functions $source(m)$ and $ts(m)$.

Process i keeps the following local variables:

- $correct_i$: the set of processes considered correct by process i;
- $last_i[n]$: an array of n elements to keep the last messages delivered by i ($last_i[j]$ is the last message broadcast by j that was delivered by i);
- ack_set_i: a set with all pending acknowledgement messages of process i. For each message $\langle TREE, m\rangle$ received by i from process j and retransmitted to process k, an element $\langle j, k, m\rangle$ is added to this set; The symbol $\perp$ represents a null element. The asterisk is used as a



(a) No failures (b) p_4 suspected

Figure A.2: Reliable broadcast - process 0 (p_0)

wildcard. For instance, $\langle j, *, m\rangle$ means all pending *acks* for a message m received from process j and re-sent to any other process;

- $pending_i$: list of the messages received by i that where not delivered yet because they are "out of order" with regard to their timestamp, i.e., $ts(m) > ts(last_i(source(m)) + 1$;
- $history_i$: the history of messages that were already broadcast by i. This set is used to prevent sending the same message to the same cluster more than once. $\langle j, m, h\rangle \in history_i$ indicates that the message m received from process j was already sent by i to the clusters $c_{i,s}$ for all $s \in [1, h]$.

A.5.2 Algorithm description

Process i broadcasts a message by calling the BROADCAST(m) function. Line 7 ensures that a new *broadcast* starts only after the previous one has been completed, i.e., there is no pending *acks* for the $last_i[i]$ message. Note that some processes might not have received the previous message yet because of false suspicions. Then, the received message m is locally delivered to i (line 9) and, by calling the function BROADCAST_TREE (line 10), i forwards m to its neighbors in the VCube. To this end, it calls, for each cluster $s \in [1, log_2 n]$, the function BROADCAST_CLUSTER that sends a `TREE` message to the first process k which is correct in the cluster (line 27). To those processes that are not correct, i.e, suspected of being crashed, and placed before k in the cluster, a `DELV` message (line 31) is sent to them. Notice that in both cases, the messages are sent provided i has not already forwarded m, received from j, to k. For every sent `TREE` message the corresponding *ack* is included in the list of pending *acks* (line 28).

Let's consider the 3-VCube topology of Figure A.2. Figure A.2(a) shows a fault-free scenario where process 0 (p_0) broadcasts a message. After delivering the message to itself, p_0 sends a copy of the message to p_1, p_2, and p_4, which are neighbors of p_0 and the first correct process on each of i's clusters.

Upon reception of a message $\langle TREE, m\rangle$ from process j (line 45), process i calls the function HANDLE_MESSAGE. In this function, m is added to the set of pending messages and then all pending messages which were broadcast by the same process that broadcast m ($source(m)$) are

Algorithm 8 Reliable broadcast - process i

1: $last_i[n] \leftarrow \{\perp, ..., \perp\}$
2: $ack\_set_i \leftarrow \emptyset$
3: $correct_i \leftarrow \{0, ..., n-1\}$
4: $pending_i \leftarrow \emptyset$
5: $history_i \leftarrow \emptyset$

6: **procedure** BROADCAST(message m)
7: **wait until** $ack\_set_i \cap \{\langle \perp, *, last_i[i] \rangle\} = \emptyset$
8: $last_i[i] \leftarrow m$
9: DELIVER(m)
10: BROADCAST_TREE($\perp, m, log_2 n$)

11: **procedure** BROADCAST_TREE(process j, message m, integer h)
12: $start \leftarrow 0$
13: **if** $\exists x : \langle j, m, x \rangle \in history_i$ **then**
14: $start \leftarrow x$
15: $history_i \leftarrow history_i \backslash \{\langle j, m, x \rangle\}$
16: $history_i \leftarrow history_i \cup \{\langle j, m, max(start, h) \rangle\}$
17: **if** $start < h$ **then**
18: **for all** $s \in [start + 1, h]$ **do**
19: BROADCAST_CLUSTER(j, m, s)

20: **procedure** BROADCAST_CLUSTER(process j, message m, integer s)
21: $sent \leftarrow false$
22: **for all** $k \in c_{i,s}$ **do**
23: **if** $sent = false$ **then**
24: **if** $\langle j, k, m \rangle \in ack\_set_i$ **and** $k \in correct_i$ **then**
25: $sent \leftarrow true$
26: **else if** $k \in correct_i$ **then**
27: SEND($\langle TREE, m \rangle$) to p_k
28: $ack\_set_i \leftarrow ack\_set_i \cup \{\langle j, k, m \rangle\}$
29: $sent \leftarrow true$
30: **else if** $\langle j, k, m \rangle \notin ack\_set_i$ **then**
31: SEND($\langle DELV, m \rangle$) to p_k

32: **procedure** CHECK_ACKS(process j, message m)
33: **if** $j \neq \perp$ **and** $ack\_set_i \cap \{\langle j, *, m \rangle\} = \emptyset$ **then**
34: SEND($\langle ACK, m \rangle$) to p_j

35: **procedure** HANDLE_MESSAGE(process j, message m)
36: $pending_i \leftarrow pending_i \cup \{m\}$
37: **while** $\exists l \in pending_i : source(l) = source(m)$
38: $\wedge (ts(l) = ts(last_i[source(l)]) + 1$
39: **or** $last_i[source(l)] = \perp \wedge ts(l) = 0)$ **do**
40: $last_i[source(l)] \leftarrow l$
41: $pending_i \leftarrow pending_i \backslash \{l\}$
42: DELIVER(l)
43: **if** $source(m) \notin correct_i$ **then**
44: BROADCAST_TREE($j, last_i[source(m)], log_2 n$)

45: **upon reception of** TREE(m) **from** p_j **do**
46: HANDLE_MESSAGE(m)
47: BROADCAST_TREE($j, m, cluster_i(j) - 1$)
48: CHECK_ACKS(j, m)

49: **upon reception of** DELV(m) **from** p_j **do**
50: HANDLE_MESSAGE(m)

51: **upon reception of** ACK(m) **from** p_j **do**
52: **for all** $k = x : \langle x, j, m \rangle \in ack\_set_i$ **do**
53: $ack\_set_i \leftarrow ack\_set_i \backslash \{\langle k, j, m \rangle\}$
54: CHECK_ACKS(k, m)

55: **when notifying** `crash`(j)
56: $correct_i \leftarrow correct_i \backslash \{j\}$
57: **for all** $p = x, m = y : \langle x, j, y \rangle \in ack\_set_i \cap \{\langle *, j, * \rangle\}$ **do**
58: BROADCAST_CLUSTER($p, m, cluster_i(j)$)
59: $ack\_set_i \leftarrow ack\_set_i \backslash \{\langle p, j, m \rangle\}$
60: CHECK_ACKS(p, m)
61: **if** $last_i[j] \neq \perp$ **then**
62: BROADCAST_TREE($j, last_i[j], log_2 n$)

63: **when notifying** `up`(j)
64: $correct_i \leftarrow correct_i \cup \{j\}$

delivered in increasing order of timestamps, provided no message is missing in the sequence of timestamps (lines 36 - 42). In the same HANDLE_MESSAGE function, if i suspects that $source(m)$ failed, it restarts the broadcast of $last_i[source(m)]$ (line 44) to ensure that every correct process receives the message even if $source(m)$ crashed in the middle of the broadcast. Otherwise, by calling the function BROADCAST_TREE with parameter $h = cluster_i(j) - 1$ (line 47), m is forwarded to all neighbors of i in each sub-cluster of i that should receive m. Figure A.2(a) shows the forwarding of m from p_4 to p_5.

If process i is a leaf in the spanning tree of the broadcast ($cluster_i(j) - 1 = 0$) or if all neighbors of i (i.e., children of i in the tree) that should receive the message are suspected of being crashed, i sends an ACK message to the process which sent m to it, by calling function CHECK_ACKS (line 34).

If process i receives a $\langle DELV, m\rangle$ message from j (line 49), it means that j falsely suspects i of being crashed and has decided to trust another process with the forwarding of the message to the rest of the tree. Therefore i can simply call the HANDLE_MESSAGE function to deliver the message and does not need to call BROADCAST_TREE.

Whenever i receives a message $\langle ACK, m\rangle$, it removes the corresponding *ack* from set of pending *acks* (line 53) and, by calling the function CHECK_ACKS, if there are no more pending *acks* for message m, i sends an ACK message to the process j which sent m to it (line 34). If $j = \perp$, the ACK message has reached the process that has broadcast m ($source(m)$) and the ACK message does not need to be forwarded.

The detection of the failure of process j is notified to i (crash(j)). It is worth pointing out that this detection might be a false suspicion. Three actions are taken by i upon receiving such a notification: (1) update of the set of processes that it considers correct (line 56); (2) removal from the set of pending *acks* of those *acks* whose related message m has been retransmitted to j (line 59); (3) re-sending to k, the next neighbor of j in the cluster of j (if k exists), of those messages previously sent to j. The re-sending of these messages triggers the propagation of messages over a new spanning tree (line 58). For instance, in Figure A.2(b), after the notification of the failure of p_4, p_0 sends message m to p_5 since the latter is the next fault-free neighbor of p_4 in $c_{0,3} = \{4, 5, 6, 7\}$ (*cluster* $s = 3$) The message is then propagated to the other correct processes of the cluster, i.e., processes p_6 and p_7. Notice that if p_4 is considered faulty by p_0 before the start of the broadcast, p_0 sends a DELV message to p_4 in order to ensure the reception and handle of m by p_4. Finally, in case of crash of j, i has to re-broadcast the last message broadcast by j (line 62). Notice that, in this case, the *history* variable is used in order to prevent i from re-rebroadcasting the message to those clusters that i has already sent the same message.

If VCube detects that it had falsely suspected process j, it corrects its mistake and notifies i which then includes j in its set of correct processes (line 64).

A.5.3 Proof of correctness

In this section we will prove that Algorithm 8 implements a reliable broadcast.

Lemma A.1. *Algorithm 8 ensures the validity property of reliable broadcast.*

Proof. If a process i broadcasts a message m, the only way that i would not deliver m is if i waits forever on line 7. This wait is interrupted when the set ack_set_i contains no more pending

acknowledgements related to the message $last_i[i]$ previously broadcast by i.

For any process j that i sent $last_i[i]$ to, i added a pending ack in ack_set_i (line 28). If j is correct, then it will eventually answer with an ACK message (line 34) and i will remove $\langle \perp, j, last_i[j] \rangle$ from ack_set_i on line 53. If j is faulty, then i will eventually detect the crash and remove the pending ack on line 59.

As a result, all of the pending acks for $last_i[i]$ will eventually be removed from ack_set_i and i will deliver m on line 9.

Line 9 then ensures that i will deliver the message before broadcasting it. □

Lemma A.2. *For any processes i and j, the value of $ts(last_i[j])$ only increases over time.*

Proof. For the sake of simplicity, we take the convention that $ts(\perp) = -1$. The $last_i$ array is only modified on lines 8 and 40.

The first case can only happen when i broadcasts a new message m, and since timestamps of new messages sent by a same processes have to be increasing, $ts(m) > ts(last_i[i])$. When i calls the *broadcast* procedure with m, $ts(last_i[i])$ will therefore increase on line 8.

The other way for $last_i$ to be modified is on line 40. $last_i[source(l)]$ will then be updated with message l if $last_i[source(l)] = \perp$ and $ts(l) = 0$ (and therefore $ts(last_i[source(l)]) = -1 < ts(l)$), or if $ts(l) = ts(last_i[source(l)]) + 1$. It follows that $last_i[source(l)$ is only updated if the new value of $ts(last_i[source(l)])$ would be superior to the old one. □

Lemma A.3. *Algorithm 8 ensures the integrity property of reliable broadcast.*

Proof. Processes only deliver a message if they are broadcasting it themselves (line 9) or if the message is in their $pending_i$ set (line 42). Messages are only added to the $pending_i$ set on line 36, after they have been received from another process. Since the links are reliable and do not create messages, it follows that a message is delivered only if it was previously broadcast (there is no creation of messages).

To show that there is no duplication of messages, let us consider two cases:

- **source(m) = i**. Process i called the *broadcast* procedure with parameter m. As proved in Lemma A.1, i will deliver m on line 9. Since the *broadcast* procedure is only called once with a given message, the only way that i would deliver m a second time is on line 42. Since $last_i[i]$ was set to m on line 8, it follows from Lemma A.2 that m will never qualify to pass the test on lines 37 – 39.

- **source(m) ≠ i**. Process i is not the emitter of message m, and did not call the *broadcast* procedure with m. Therefore the only way for i to deliver m is on line 42. Before i delivers m for the first time, it sets $last_i[source(m)]$ to m on line 40. It then follows from Lemma A.2 that m will never again qualify to pass the test on lines 37 – 39, and therefore i can deliver m at most once.

□

Lemma A.4. *Algorithm 8 ensures the agreement property of reliable broadcast.*

Proof. Let m be a message broadcast by a process i. We consider two cases:

- **i is correct**. It can be shown by induction that every correct process receives m.

  As a basis of the induction, let us consider the case where $n = 2$ and $P = \{i, j\}$. It follows that $c_{i,1} = \{j\}$. Therefore i will send m to j on line 31 if i suspects j or on line 27 otherwise. If j is correct, it will eventually receive m since the links are reliable, and will deliver m on line 42. i will also deliver m, by virtue of the validity property.

  We now have to prove that if every correct process receives m for $n = 2^k$, it is also the case for $n = 2^{k+1}$. The system of size 2^{k+1} can be seen as two subsystems $P_1 = \{i\} \cup \bigcup_{x=1}^{k} c_{i,x}$ and $P_2 = c_{i,k+1}$ such that $|P_1| = |P_2| = 2^k$.

  The *broadcast_tree* and *broadcast_cluster* procedures ensure that for every $s \in [1, k+1]$, i will send m to at least one process in $c_{i,s}$. Let j be the first process in $c_{i,k+1}$. If j is correct, it will eventually receive m. If j is faulty and i detected the crash prior to the broadcast, i will send the message to j anyway in case it is a false suspicion (line 31) but it will also send it to another process in $c_{i,k+1}$ as a precaution (line 27). i will keep doing so until it has sent the $TREE$ message to a non-suspected process in $c_{i,k+1}$, or until it has sent the message to all the processes in $c_{i,k+1}$.

  If j is faulty and i only detects the crash after the broadcast, the *broadcast_cluster* procedure will be called again on line 58, which ensures once again that i will send the message to a non-suspected process in $c_{i,k+1}$. As a result, unless all the processes in $c_{i,k+1}$ are faulty, at least one correct process in $c_{i,k+1}$ will eventually receive m. This correct process will then broadcast m to the rest of the P_2 subsystem on line 47.

  Since a correct process broadcasts m in both subsystems P_1 and P_2, and since both subsystems are of size 2^k, it follows that every correct process in P will eventually receive m.

- **i is faulty**. If i crashes before sending m to any process, then no correct process delivers m and the agreement property is verified. If i crashes after the broadcast is done, then everything happens as if i was correct. If i crashes after sending m to some processes and a correct process j receives m, then j will eventually detect the failure of i. If j detects the crash before receiving m, when it receives m it will restart a full broadcast of m on line 44. If j only detects the crash of i after receiving m, it will also restart a full broadcast of m on line 62. Since j is correct, every correct process will eventually receive m.

□

Theorem A.1. *Algorithm 8 implements a reliable broadcast.*

Proof. The proof follows directly from Lemmas A.1, A.3, and A.4. □

A.6 Performance Discussion

The goal of exploiting the VCube overlay in our solution is to provide an efficient broadcast where each process sends at most $log_2 n$ messages. However, this complexity cannot be ensured at all times in an asynchronous system where false suspicions can arise. Algorithm 8 aims to

take advantage of the VCube whenever possible while still ensuring the properties of a reliable broadcast despite false suspicions.

In the best case scenario where no process is ever suspected of failure, each process will send at most one message per cluster (line 27). Therefore $n-1$ $TREE$ messages will be sent in total (since no process will be sent the same message twice) with no single process sending more than $log_2 n$ messages. This is the example presented in Figure A.2(a).

If a process other than the source of the broadcast is suspected before the broadcast, there will be $n-2$ $TREE$ messages and one $DELV$ message sent. A single process might send up to $log_2 n$ $TREE$ messages plus one $DELV$ message per suspected process. This is the example of Figure A.2(b).

If the source of the broadcast suspects everyone else, then it will send $n-1$ $DELV$ messages. In this case, Algorithm 8 is equivalent to a *one-to-all* algorithm where one process sends the message directly to all others, losing, thus, the advantages of tree topology properties, such as scalability.

The main cost of suspicions lies in the fact that when a process is suspected, its last broadcast must be resent. This is the purpose of lines 44 and 62. Such a re-broadcast is an unavoidable consequence of the existence of false suspicions, necessary in order to ensure the agreement property of reliable broadcast.

Note that the fact that the information about a node failure is false or true has no difference in the impact on the performance of the broadcast algorithm in terms of message complexity.

A.7 Conclusion and Future Work

This article presented a reliable broadcast algorithm for message-passing distributed systems prone to crash failures on asynchronous environments. It tolerates up to n-1 failures. For broadcasting a message, the algorithm dynamically builds a spanning tree over a virtual hypercube topology provided by the underlying monitor system VCube. In case of failure, the tree is dynamically reconstructed. To this end, the VCube provides information about node failures. However, as the system is asynchronous, it can make mistake falsely suspecting no faulty nodes. Such false suspicions are tolerated by the algorithm by sending special messages to those processes suspected of having failed. In summary, whenever possible, the algorithm exploits the hypercube properties offered by the VCube while ensuring the properties of the reliable broadcast, even in case of false suspicions.

As future work, we intend to implement our algorithm and conduct extensive simulation experiments in order to compare its performance in terms of latency and number of messages in different scenarios with and without failure of nodes as well as false suspicions.

Bibliography

[ABD95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-Passing Systems. *J. ACM*, 42(1):124–142, 1995.

[ACT00] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[ADGF08] Mohssen Abboud, Carole Delporte-Gallet, and Hugues Fauconnier. Agreement Without Knowing Everybody: A First Step to Dynamicity. NOTERE '08, pages 49:1–49:5. ACM, 2008.

[AEA91] Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 9(1):1–20, February 1991.

[AG13] Yehuda Afek and Eli Gafni. Asynchrony from synchrony. In *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, volume 7730 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2013.

[AGSS13] Luciana Arantes, Fabíola Greve, Pierre Sens, and Véronique Simon. Eventual leader election in evolving mobile networks. In *Principles of Distributed Systems - 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings*, volume 8304, pages 23–37, 2013.

[Agu04] Marcos Kawazoe Aguilera. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59, 2004.

[ASSC02] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[AST+10] Luciana Arantes, Pierre Sens, Gaël Thomas, Denis Conan, and Léon Lim. Partition participant detector with dynamic paths in mobile networks. In *Proceedings of The Ninth IEEE International Symposium on Networking Computing and Applications, NCA 2010, July 15-17, 2010, Cambridge, Massachusetts, USA*, pages 224–228, 2010.

[AWW05] Ian F. Akyildiz, Xudong Wang, and Weilin Wang. Wireless mesh networks: a survey. *Computer Networks*, 47(4):445–487, 2005.

[BBRT07] Roberto Baldoni, Marin Bertier, Michel Raynal, and Sara Tucci Piergiovanni. Looking for a definition of dynamic distributed systems. In *Parallel Computing Technologies, 9th International Conference, PaCT 2007, Pereslavl-Zalessky, Russia, September 3-7, 2007, Proceedings*, pages 1–14, 2007.

[BCJ09] Vibhor Bhatt, Nicholas Christman, and Prasad Jayanti. Extracting quorum failure detectors. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 73–82, 2009.

[BDPB13] Silvia Bonomi, Antonella Del Pozzo, and Roberto Baldoni. Intrusion-tolerant reliable broadcast. Technical report, Sapienza Università di Roma,, 2013.

[BF03] Sandeep Bhadra and Afonso Ferreira. Complexity of connected components in evolving graphs and the computation of multicast trees in dynamic networks. In *Ad-Hoc, Mobile, and Wireless Networks, Second International Conference, ADHOC-NOW 2003 Montreal, Canada, October 8-10, 2003, Proceedings*, pages 259–270, 2003.

[BFJ03] Binh-Minh Bui-Xuan, Afonso Ferreira, and Aubin Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. Found. Comput. Sci.*, 14(2):267–285, 2003.

[BG93] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 91–100, 1993.

[BLG15] Abdulkader Benchi, Pascale Launay, and Frédéric Guidec. Solving consensus in opportunistic networks. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, pages 1:1–1:10, 2015.

[BR09] François Bonnet and Michel Raynal. Looking for the Weakest Failure Detector for k-Set Agreement in Message-Passing Systems: Is Π_k the End of the Road? In *Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009. Proceedings*, volume 5873, pages 149–164, 2009.

[BRS09] Martin Biely, Peter Robinson, and Ulrich Schmid. Weak synchrony models and failure detectors for message passing (k-)set agreement. In *Principles of Distributed Systems, 13th International Conference, OPODIS 2009, Nîmes, France, December 15-18, 2009. Proceedings*, pages 285–299, 2009.

[BRS11] Martin Biely, Peter Robinson, and Ulrich Schmid. Easy impossibility proofs for k-set agreement in message passing systems. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 227–228, 2011.

[BRS12] Martin Biely, Peter Robinson, and Ulrich Schmid. Agreement in directed dynamic networks. In *Structural Information and Communication Complexity - 19th International Colloquium, SIROCCO 2012, Reykjavik, Iceland, June 30-July 2, 2012, Revised Selected Papers*, volume 7355, pages 73–84, 2012.

[BRS14] Martin Biely, Peter Robinson, and Ulrich Schmid. The generalized loneliness detector and weak system models for k-set agreement. *IEEE Trans. Parallel Distrib. Syst.*, 25(4):1078–1088, 2014.

[BRS+18] Martin Biely, Peter Robinson, Ulrich Schmid, Manfred Schwarz, and Kyrill Winkler. Gracefully degrading consensus and k-set agreement in directed dynamic networks. *Theor. Comput. Sci.*, 726:41–77, 2018.

[BT10] Zohir Bouzid and Corentin Travers. (anti-$\Omega^x \times \Sigma_z$)-Based k-Set Agreement Algorithms. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, volume 6490, pages 189–204, 2010.

[CCF11] Arnaud Casteigts, Serge Chaumette, and Afonso Ferreira. On the Assumptions about Network Dynamics in Distributed Computing. *CoRR*, abs/1102.5529, 2011.

[CFG+15] Arnaud Casteigts, Paola Flocchini, Emmanuel Godard, Nicola Santoro, and Masafumi Yamashita. On the expressivity of time-varying graphs. *Theor. Comput. Sci.*, 590:27–37, 2015.

[CFQS12] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *IJPEDS*, 27(5):387–408, 2012.

[CG13] Étienne Coulouma and Emmanuel Godard. A characterization of dynamic networks where consensus is solvable. In *Structural Information and Communication Complexity - 20th International Colloquium, SIROCCO 2013, Ischia, Italy, July 1-3, 2013, Revised Selected Papers*, pages 24–35, 2013.

[Cha93] Soma Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Inf. Comput.*, 105(1):132–158, 1993.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *JACM*, 43(4):685–722, 1996.

[CRTW07] Jiannong Cao, Michel Raynal, Corentin Travers, and Weigang Wu. The eventual leadership in dynamic mobile networking environments. In *13th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2007), 17-19 December, 2007, Melbourne, Victoria, Australia*, pages 123–130, 2007.

[CSL90] Ye-In Chang, Mukesh Singhal, and Ming T. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *Ninth Symposium on Reliable Distributed Systems, SRDS 1990, Huntsville, Alabama, USA, October 9-11, 1990, Proceedings*, pages 146–154, 1990.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *JACM*, 43(2):225–267, 1996.

[CZCL07] Wei Chen, Jialin Zhang, Yu Chen, and Xuezheng Liu. Weakening failure detectors for k -set agreement via the partition approach. In *Distributed Computing, 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007, Proceedings*, pages 123–138, 2007.

[DBR14] Elias P. Duarte, Jr., Luis C. E. Bona, and Vinicius K. Ruoso. VCube: A provably scalable distributed diagnosis algorithm. In *5th Work. on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA'14, pages 17–22, Piscataway, USA, 2014. IEEE Press.

[DFG08] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Sharing is harder than agreeing. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 85–94, 2008.

[DFG10] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *JACM*, 57(4), 2010.

[DFGT08] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Andreas Tielmann. The weakest failure detector for message passing set-agreement. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, pages 109–120, 2008.

[DGFGK05] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65(4):492–505, apr 2005.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[DJN98] E. P. Duarte Jr. and T. Nanya. A hierarchical adaptive distributed system-level diagnosis algorithm. *IEEE Trans. Comput.*, 47(1):34–45, January 1998.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[FA96] P. Fragopoulou and S.G. Akl. Edge-disjoint spanning trees on the star network with applications to fault tolerance. *IEEE Trans. Comput.*, 45(2):174 –185, February 1996.

[FGA11] Paulo Floriano, Alfredo Goldman, and Luciana Arantes. Formalization of the necessary and sufficient connectivity conditions to the distributed mutual exclusion problem in dynamic networks. In *Proceedings of The Tenth IEEE International Symposium on Networking Computing and Applications, NCA 2011, August 25-27, 2011, Cambridge, Massachusetts, USA*, pages 203–210, 2011.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *JACM*, 32(2):374–382, 1985.

[FLR10] Mário F. S. Ferreira, João Leitão, and Luís E. T. Rodrigues. Thicket: A protocol for building and maintaining multiple trees in a P2P overlay. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*, pages 293–302, 2010.

[FMR05] Roy Friedman, Achour Mostéfaoui, and Michel Raynal. Asynchronous bounded lifetime failure detectors. *Inf. Process. Lett.*, 94(2):85–91, 2005.

[FT09] Roy Friedman and Galya Tcharny. Evaluating failure detection in mobile ad-hoc networks. *Int. J. Pervasive Computing and Communications*, 5(4):476–496, 2009.

[GAS11] Fabíola Greve, Luciana Arantes, and Pierre Sens. What model and what conditions to implement unreliable failure detectors in dynamic networks? In *Workshop on Theoretical Aspects on Dynamic Distributed Systems, TADDS '11, Rome, Italy, September 19, 2011*, pages 13–17, 2011.

[GCLL15] Carlos Gómez-Calzado, Arnaud Casteigts, Alberto Lafuente, and Mikel Larrea. A connectivity model for agreement in dynamic systems. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, pages 333–345, 2015.

[GdLAS12] Fabíola Greve, Murilo Santos de Lima, Luciana Arantes, and Pierre Sens. A time-free byzantine failure detector for dynamic networks. In *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, pages 191–202, 2012.

[GH17] Wojciech M. Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 211–220, 2017.

[GHK+07] Rachid Guerraoui, Maurice Herlihy, Petr Kouznetsov, Nancy A. Lynch, and Calvin C. Newport. On the weakest failure detector ever. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 235–243, 2007.

[GK09] Eli Gafni and Petr Kuznetsov. The weakest failure detector for solving k-set agreement. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 83–91, 2009.

[GLLR13] Carlos Gómez-Calzado, Alberto Lafuente, Mikel Larrea, and Michel Raynal. Fault-tolerant leader election in mobile dynamic distributed systems. In *IEEE 19th Pacific Rim International Symposium on Dependable Computing, PRDC 2013, Vancouver, BC, Canada, December 2-4, 2013*, pages 78–87, 2013.

[GR06] Rachid Guerraoui and Luís E. T. Rodrigues. *Introduction to reliable distributed programming.* Springer, 2006.

[GR07] Rachid Guerraoui and Michel Raynal. The Alpha of Indulgent Consensus. *Comput. J.*, 50(1):53–67, 2007.

[GR16] Wojciech M. Golab and Aditya Ramaraju. Recoverable mutual exclusion: [extended abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 65–74, 2016.

[GSAS11] Fabíola Greve, Pierre Sens, Luciana Arantes, and Véronique Simon. A failure detector for wireless networks with unknown membership. In *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*, volume 6853 of *Lecture Notes in Computer Science*, pages 27–38. Springer, 2011.

[GSAS12] Fabíola Greve, Pierre Sens, Luciana Arantes, and Véronique Simon. Eventually Strong Failure Detector with Unknown Membership. *Comput. J.*, 55(12):1507–1524, 2012.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. chapter Distributed systems, pages 97–145. ACM Press, New York, NY, USA, 2 edition, 1993.

[JAF06] Ernesto Jiménez, Sergio Arévalo, and Antonio Fernández. Implementing unreliable failure detectors with unknown membership. *Inf. Process. Lett.*, 100(2):60–63, 2006.

[JJ17] Prasad Jayanti and Anup Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 30:1–30:15, 2017.

[JRAJ16] Denis Jeanneau, Luiz A. Rodrigues, Luciana Arantes, and Elias Procópio Duarte Jr. An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detector. In *2016 Seventh Latin-American Symposium on Dependable Computing, LADC 2016, Cali, Colombia, October 19-21, 2016*, pages 91–98, 2016.

[JRAJ17] Denis Jeanneau, Luiz A. Rodrigues, Luciana Arantes, and Elias Procópio Duarte Jr. An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection. *J. Braz. Comp. Soc.*, 23(1):15:1–15:14, 2017.

[JRAS15] Denis Jeanneau, Thibault Rieutord, Luciana Arantes, and Pierre Sens. A failure detector for k-set agreement in dynamic systems. In *14th IEEE International Symposium on Network Computing and Applications, NCA 2015, Cambridge, MA, USA, September 28-30, 2015*, pages 176–183, 2015.

[JRAS17] Denis Jeanneau, Thibault Rieutord, Luciana Arantes, and Pierre Sens. Solving k-set agreement using failure detectors in unknown dynamic networks. *IEEE Trans. Parallel Distrib. Syst.*, 28(5):1484–1499, 2017.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[KLO10] Fabian Kuhn, Nancy A. Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 513–522. ACM, 2010.

[KMO11] Fabian Kuhn, Yoram Moses, and Rotem Oshman. Coordinated consensus in dynamic networks. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 1–10, 2011.

[KMV10] Kyungbaek Kim, Sharad Mehrotra, and Nalini Venkatasubramanian. FaReCast: Fast, reliable application layer multicast for flash dissemination. In *ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware'10, pages 169–190, Berlin, Heidelberg, 2010. Springer-Verlag.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[Lam86] Leslie Lamport. The mutual exclusion problem: part I&II. *J. ACM*, 33(2):313–348, 1986.

[Lam98] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[LB99] Jörg Liebeherr and Tyler K. Beam. Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Networked Group Communication, First International COST264 Workshop, NGC'99, Pisa, Italy, November 17-20, 1999, Proceedings*, pages 72–89. 1999.

[LMS01] Igor Litovsky, Yves Métivier, and Eric Sopena. Graph Relabelling Systems and Distributed Algorithms. In *Handbook of graph grammars and computing by graph transformation*, pages 1–56. World scientific, 2001.

[LPR07] João Leitão, José Pereira, and Luís E. T. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pages 419–429, 2007.

[LRAC12] Mikel Larrea, Michel Raynal, Iratxe Soraluze Arriola, and Roberto Cortiñas. Specifying and implementing an eventual leader service for dynamic systems. *IJWGS*, 8(3):204–224, 2012.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[MJAS19] Etienne Mauffret, Denis Jeanneau, Luciana Arantes, and Pierre Sens. The weakest failure detector to solve the mutual exclusion problem in an unknown dynamic environment. In *Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 4-7, 2019 (to be published)*, 2019.

[MMR03] Achour Mostéfaoui, Eric Mourgaya, and Michel Raynal. Asynchronous implementation of failure detectors. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, pages 351–360, 2003.

[MR99] Achour Mostéfaoui and Michel Raynal. Unreliable failure detectors with limited scope accuracy and an application to consensus. In *Foundations of Software Technology and Theoretical Computer Science, 19th Conference, Chennai, India, December 13-15, 1999, Proceedings*, pages 329–340, 1999.

[MR00] Achour Mostéfaoui and Michel Raynal. k-set agreement with limited accuracy failure detectors. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.*, pages 143–152, 2000.

[MRS11] Achour Mostéfaoui, Michel Raynal, and Julien Stainer. Relations linking failure detectors associated with k-set agreement in message-passing systems. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, pages 341–355, 2011.

[MRS12] Achour Mostéfaoui, Michel Raynal, and Julien Stainer. Chasing the weakest failure detector for k-set agreement in message-passing systems. In *11th IEEE International Symposium on Network Computing and Applications, NCA 2012, Cambridge, MA, USA, August 23-25, 2012*, pages 44–51, 2012.

[MRT+05] Achour Mostéfaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and Amr El Abbadi. From static distributed systems to dynamic systems. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005),26-28 October 2005, Orlando, FL, USA*, pages 109–118, 2005.

[Nei95] Gil Neiger. Failure detectors and the wait-free hierarchy. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 100–109, 1995.

[NLM90] Shojiro Nishio, Kin F. Li, and Eric G. Manning. A Resilient Mutual Exclusion Algorithm for Computer Networks. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):344–355, 1990.

[RAJ14] Luiz A. Rodrigues, Luciana Arantes, and Elias Procópio Duarte Jr. An autonomic implementation of reliable broadcast based on dynamic spanning trees. In *2014 Tenth European Dependable Computing Conference, Newcastle, United Kingdom, May 13-16, 2014*, pages 1–12, 2014.

[RAS15] Thibault Rieutord, Luciana Arantes, and Pierre Sens. Détecteur de défaillances minimal pour le consensus adapté aux réseaux inconnus. In *Algotel*, 2015.

[Ray86] Michel Raynal. *Algorithms for Mutual Exclusion.* MIT Press, Cambridge, MA, USA, 1986.

[Ray07] Michel Raynal. K-anti-omega. In *Rump Session at 26th ACM Symposium on Principles of Distributed Computing (PODC'07)*, 2007.

[Ray10] Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[Ray11] Michel Raynal. Failure detectors to solve asynchronous k-set agreement: a glimpse of recent results. *Bulletin of the EATCS*, 103:74–95, 2011.

[RS88] P. Ramanathan and K.G. Shin. Reliable broadcast in hypercube multicomputers. *IEEE Trans. Comput.*, 37(12):1654–1657, 1988.

[RSCW14] Michel Raynal, Julien Stainer, Jiannong Cao, and Weigang Wu. A simple broadcast algorithm for recurrent dynamic systems. In *28th IEEE International Conference on Advanced Information Networking and Applications, AINA 2014, Victoria, BC, Canada, May 13-16, 2014*, pages 933–939, 2014.

[RT06] Michel Raynal and Corentin Travers. In search of the holy grail: Looking for the weakest failure detector for wait-free set agreement. In *Principles of Distributed Systems, 10th International Conference, OPODIS 2006, Bordeaux, France, December 12-15, 2006, Proceedings*, volume 4305, pages 3–19, 2006.

[SAB+08] Pierre Sens, Luciana Arantes, Mathieu Bouillaguet, Véronique Simon, and Fabíola Greve. An unreliable failure detector for unknown and mobile networks. In *Principles of Distributed Systems, 12th International Conference, OPODIS 2008, Luxor, Egypt, December 15-18, 2008. Proceedings*, pages 555–559, 2008.

[SAS06] Julien Sopena, Luciana Bezerra Arantes, and Pierre Sens. Performance evaluation of a fair fault-tolerant mutual exclusion algorithm. In *25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006),2-4 October 2006, Leeds, UK*, pages 225–234, 2006.

[SGS84] Fred B. Schneider, David Gries, and Richard D. Schlichting. Fault-tolerant broadcasts. *Sci. Comput. Program.*, 4(1):1–15, 1984.

[SS14] Adam Sealfon and Aikaterini A. Sotiraki. Agreement in partitioned dynamic networks. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 555–556. Springer, 2014.

[TI15] Erfan Taheri and Mohammad Izadi. Byzantine consensus for unknown dynamic networks. *The Journal of Supercomputing*, 71(4):1587–1603, 2015.

[Wu96] Jie Wu. Optimal broadcasting in hypercubes with link faults using limited global information. *J. Syst. Archit.*, 42(5):367–380, 1996.

[Zie08] Piotr Zielinski. Anti-omega: the weakest failure detector for set agreement. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 55–64, 2008.