Tolerating Transient, Permanent, and Intermittent Failures

Swan Dubois

► **To cite this version:**

Swan Dubois. Tolerating Transient, Permanent, and Intermittent Failures. Distributed, Parallel, and Cluster Computing [cs.DC]. Université Pierre et Marie Curie - Paris VI, 2011. English. NNT : . tel-00663317

HAL Id: tel-00663317

https://theses.hal.science/tel-00663317

Submitted on 26 Jan 2012

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

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

UNIVERSITÉ PIERRE ET MARIE CURIE – PARIS 6

ÉCOLE DOCTORALE EDITE

ÉCOLE DOCTORALE INFORMATIQUE, TÉLÉCOMMUNICATIONS ET ÉLECTRONIQUE

THÈSE

pour obtenir le grade de

Docteur en Sciences

de l'Université Pierre et Marie Curie – Paris 6

Mention : Systèmes Informatiques

Présentée et soutenue par

Swan Dubois

Tolérer les fautes transitoires, permanentes et intermittentes

Thèse dirigée par Sébastien Tixeuil
et co-encadrée par Maria Potop-Butucaru

préparée au Laboratoire d'Informatique de Paris 6 (LIP6),
équipe-projet INRIA REGAL,

soutenue publiquement le 1er décembre 2011,

après avis des rapporteurs :

Bertrand Ducourthial	–	Professeur, UTC
Ted Herman	–	Professeur, Université de l'Iowa

et devant le jury composé de :

Rapporteur	Bertrand Ducourthial	–	Professeur, UTC
Examinateurs	Carole Delporte-Gallet	–	Professeur, Université Paris 7
	Rachid Guerraoui	–	Professeur, EPFL
	Nicolas Hanusse	–	Directeur de Recherche, CNRS
	Pierre Sens	–	Professeur, UPMC
Directeur	Sébastien Tixeuil	–	Professeur, UPMC
Co-encadrant	Maria Potop-butucaru	–	MCF, HDR, UPMC

UNIVERSITÉ PIERRE ET MARIE CURIE – PARIS 6

ÉCOLE DOCTORALE EDITE

ÉCOLE DOCTORALE INFORMATIQUE,
TÉLÉCOMMUNICATIONS ET ÉLECTRONIQUE

T H È S E

pour obtenir le grade de

Docteur en Sciences

de l'Université Pierre et Marie Curie – Paris 6

Mention : Systèmes Informatiques

Présentée et soutenue par

Swan Dubois

Tolerating Transient, Permanent, and Intermittent Failures

Thèse dirigée par Sébastien Tixeuil
et co-encadrée par Maria Potop-Butucaru

préparée au Laboratoire d'Informatique de Paris 6 (LIP6),
équipe-projet INRIA REGAL,

soutenue publiquement le 1er décembre 2011,

après avis des rapporteurs :

Bertrand Ducourthial	–	Professeur, UTC
Ted Herman	–	Professeur, Université de l'Iowa

et devant le jury composé de :

Rapporteur	Bertrand Ducourthial	–	Professeur, UTC
Examinateurs	Carole Delporte-Gallet	–	Professeur, Université Paris 7
	Rachid Guerraoui	–	Professeur, EPFL
	Nicolas Hanusse	–	Directeur de Recherche, CNRS
	Pierre Sens	–	Professeur, UPMC
Directeur	Sébastien Tixeuil	–	Professeur, UPMC
Co-encadrant	Maria Potop-butucaru	–	MCF, HDR, UPMC

Computer Science is no more about computers than astronomy is about telescopes.

Edsger W. Dijkstra

Résumé

Un système réparti est un système constitué d'un ensemble d'unités de calcul autonomes dotées de capacités de communication afin de résoudre une tâche globale. Ce modèle est suffisament général pour décrire tout type de réseau physique (réseau local, réseau de capteurs, ...). Lorsque la taille d'un système réparti devient importante ou lorsque ce système est déployé dans un environnement non contrôlé, la probabilité que certains éléments du système subissent des fautes (panne, corruption de mémoire, piratage, ...) devient non négligeable. Ces fautes peuvent être classifiées en fonction de leur durée, de leur étendue et de leur nature. Dans cette thèse, nous nous intéressons aux systèmes répartis capables de tolérer simultanément plusieurs types de fautes à travers l'étude de trois problèmes fondamentaux. Nous présentons ainsi un protocole réparti simulant un registre atomique mono-écrivan multi-lecteurs en présence de fautes transitoires et de fautes permanentes de type crash. Ce protocole repose sur deux outils ré-utilisables : un protocole de communication et un système d'estampillage borné. Ensuite, nous proposons une étude de la synchronisation faible d'horloges logiques en présence de fautes transitoires et de fautes intermittentes Byzantines. Nous prouvons de nombreux résultats d'impossibilité et nous fournissons un protocole optimal dans les cas non couverts par ces résultats. Finalement, nous définissons trois nouveaux concepts de tolérance pour les systèmes répartis sujets à des fautes transitoires et des fautes intermittentes Byzantines. Nous donnons un protocole de construction d'une vaste classe d'arbres couvrants optimal selon ces trois concepts.

Abstract

A distributed system is a system composed of a set of autonomous computation units endowed with communication abilities in order to solve a global task. This model is general enough to describe any kind of network (LAN, sensor network, ...). When the size of a distributed system gets larger or when it is deployed in hazardous environments, the possibility that some elements of the system are subject to faults (failure, memory corruption, hacking, ...) become impossible to elude. Faults can be classified according to duration, span, or nature. In this thesis, we focus on distributed systems that simultaneously tolerate several kinds of faults using three classical problems as case studies. We present first a distributed protocol simulating a single-writer multi-reader atomic register in the presence of transient faults and of permanent crash faults. This protocol relies on two re-usable tools: a communication primitive and a bounded timestamp scheme. Then, we study logical clock weak synchronization in the presence of transient faults and of intermittent Byzantine faults. We prove several impossibility results and provide a protocol that is optimal both with respect to impossibility result and with respect to recovery time. Finally, we define three new fault tolerance schemes in distributed systems that are subject to transient faults and to intermittent Byzantine faults. We design a protocol constructing a wide class of spanning trees that is optimal with respect to fault tolerance metrics defined for these three schemes.

Contents

1 Introduction **1**
- 1.1 Context of the Thesis 2
- 1.2 Thesis Contributions 4

I Context **7**

2 Model **9**
- 2.1 Distributed System 9
  - 2.1.1 Characteristics 10
  - 2.1.2 Advantages 10
- 2.2 Communication Graph 11
- 2.3 Models Used in this Thesis 13
  - 2.3.1 Generic Model 13
  - 2.3.2 State Model 15
  - 2.3.3 Message Passing Model 16
- 2.4 Fault Taxonomy 18
  - 2.4.1 Faults 18
  - 2.4.2 Fault Patterns 21

3 Taxonomy of Daemons **27**
- 3.1 Characterization of Daemons 30
  - 3.1.1 Distribution 30
  - 3.1.2 Fairness 31
  - 3.1.3 Boundedness 35
  - 3.1.4 Enabledness 37
- 3.2 Comparing Daemons 41
  - 3.2.1 Comparing daemon classes 42
  - 3.2.2 Preserving execution properties 44
  - 3.2.3 The Case of the Synchronous Daemon 48
  - 3.2.4 A map of classical daemons 50
- 3.3 Daemon Transformers 51

4 Fault Tolerance **57**
- 4.1 Tolerating Transient Fault Patterns 59

4.1.1 Weakening Self-Stabilization 60
4.1.2 Enhancing Self-Stabilization 61
4.2 Tolerating Composite Fault Patterns 62
4.2.1 Fault-Tolerant Self-Stabilization 62
4.2.2 Byzantine Tolerant Self-Stabilization 63
4.2.3 Strict Stabilization 64
4.3 Summary 66

II Atomic Register 69

5 Introduction of Part II 71
5.1 Problem and Related Works 72
5.1.1 Problem 72
5.1.2 Related Works 74
5.1.3 Specification 76
5.2 Contributions of Part II 77

6 Preliminaries 79
6.1 Data-Link Protocol 79
6.1.1 Problem and Related Works 80
6.1.2 Specification 81
6.1.3 Lower Bounds 83
6.1.4 Optimal Solution 85
6.1.5 Correctness Proof 88
6.2 Bounded Labelling Systems 93
6.2.1 Problem and Related Works 93
6.2.2 Solution 96

7 Atomic Register Simulation 101
7.1 The ABD Simulation 101
7.2 The FTPS Simulation 103
7.2.1 Distributed Protocol 104
7.2.2 Proof of Correctness 106
7.2.3 Conclusion 111

8 Conclusion of Part II 113
8.1 Summary of Contributions 113
8.2 Concluding Remarks 114

III Unison 117

9 Introduction of Part III 119
9.1 Problem and Related Works 120

9.1.1 Problem 120
9.1.2 Related Works 120
9.1.3 Specification and Definitions 121
9.2 Contributions of Part III 124
9.3 Fault-Tolerant Self-Stabilization 125

10 Impossibility Results 127
10.1 General Results 129
10.1.1 Two and more Byzantine Faults 129
10.1.2 Unfair Daemon 129
10.2 Minimal Unison Related Results 130
10.2.1 Weakly Fair Daemon 130
10.2.2 Strongly Fair Daemon and Maximal Degree greater than 3 . . 133
10.3 Priority Unison Related Results 136
10.3.1 Weakly Fair Daemon 137
10.3.2 Strongly Fair Daemon and Maximal Degree greater than 3 . . 138
10.4 Summary of Impossibility Results 140

11 Strictly Stabilizing Solution 141
11.1 Strictly Stabilizing Solution 141
11.1.1 Distributed Protocol Description 142
11.1.2 Correctness Proof 143
11.2 Optimality of Convergence Time 149
11.2.1 Upper bound 149
11.2.2 Lower Bound 150
11.2.3 Conclusion 155

12 Conclusion of Part III 157
12.1 Summary of Contributions 157
12.2 Concluding Remarks 158

IV Spanning Tree 161

13 Introduction of Part IV 163
13.1 Problem and Related Works 164
13.1.1 Related Works 164
13.1.2 Specification 166
13.2 Contributions of Part IV 170
13.3 Containing Byzantine Faults in Self-Stabilization 172
13.3.1 Strict Stabilization 172
13.3.2 Strong Stabilization 173
13.3.3 Topology-Aware Stabilization 175
13.3.4 Discussion 177

14 Two Case Studies **179**
14.1 Spanning Tree without Constraints . . . 180
14.1.1 Strongly Stabilizing Distributed Protocol . . . 181
14.1.2 Proof of Strong Stabilization . . . 182
14.2 BFS Spanning Tree . . . 185
14.2.1 Impossibility of Strong Stabilization . . . 186
14.2.2 Topology-Aware Stabilizing Solution . . . 187
14.2.3 Proof of Topology-Aware Strict Stabilization . . . 188
14.2.4 Proof of Topology-Aware Strong Stabilization . . . 192
14.3 Summary . . . 195

15 General Case **197**
15.1 Topology-Aware Stabilizing Solution . . . 200
15.1.1 Distributed Protocol . . . 201
15.1.2 Proof of Topology-Aware Strict Stabilization . . . 204
15.1.3 Proof of Topology-Aware Strong Stabilization . . . 211
15.2 Optimality of Containment Areas . . . 215
15.2.1 Topology-Aware Strict Stabilization . . . 215
15.2.2 Topology-Aware Strong Stabilization . . . 217
15.3 Strong Stabilization . . . 219
15.4 Summary . . . 225

16 Conclusion of Part IV **227**
16.1 Summary of Contributions . . . 227
16.2 Concluding Remarks . . . 229

17 Conclusion **231**
17.1 Overview of Thesis Contributions . . . 231
17.1.1 Part One: Context . . . 231
17.1.2 Part Two: Atomic Register . . . 232
17.1.3 Part Three: Unison . . . 233
17.1.4 Part Four: Spanning Tree . . . 233
17.1.5 Summary . . . 234
17.2 Perspectives . . . 234

A Version française **239**
A.1 Contexte de la thèse . . . 240
A.1.1 Généralités . . . 240
A.1.2 Modèles et tolérance aux fautes . . . 243
A.2 Registre atomique . . . 245
A.2.1 Contexte . . . 245
A.2.2 Contributions . . . 246
A.2.3 Perspectives . . . 246
A.3 Unisson . . . 247

A.3.1 Contexte 247
A.3.2 Contributions 248
A.3.3 Perspectives 248
A.4 Arbre couvrant 249
A.4.1 Contexte 249
A.4.2 Contributions 250
A.4.3 Perspectives 251
A.5 Conclusion 251

Index **255**

List of Notations **257**

Bibliography **259**

List of Figures

1.1 Dependencies between chapters of this thesis. 6

2.1 Some classical communication graphs: (i) a chain, (ii) a ring, (iii) a tree, and (iv) a complete communication graph. 12

2.2 Illustration of Definition 2.5. 19

2.3 Given two actions α_1 and α_2, we have $\alpha_1 \otimes \alpha_2 = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}$ (each number represents the value of a member of the system). . . . 22

2.4 Fault patterns $FP = \{f_1, f_2, f_3\}$ and $FP' = \{f'_1, f'_2, f'_3, f'_4\}$ describe the same set of faults if all considered faults have the same behavior. 23

3.1 Mutual exclusion *vs.* asynchronous scheduling 28

3.2 Vertex coloring *vs.* synchronous scheduling 29

3.3 Inclusions of sets of daemons with respect to distribution. 31

3.4 Inclusions of sets of daemons with respect to fairness. 32

3.5 Inclusions of sets of daemons with respect to boundedness. 36

3.6 Inclusions of sets of daemons with respect to enabledness. 38

3.7 Relationship between classical daemons (an arrow from a daemon d to a daemon d' means that $d' \preccurlyeq d$, note that we remove all arrows obtained by transitivity). 51

3.8 Summary of existing daemon transformers. An arrow from a daemon to another one means that the related work provides a transformer from the first to the second. 53

4.1 Summary of respective constraints on permanent or intermittent fault tolerance schemes in self-stabilization. An arrow from a scheme to another means that the first is more constrained than the second. Note that we remove all arrows deductible from transitivity. 68

5.1 Illustartion of concurrent or consecutive operations. 73

5.2 If r_2 returns the value written by w_1 and r_1 returns the value written by w_2, we have a new/old inversion. 74

6.1 General organization of our data-link distributed protocol. 88

6.2 An example of the bounded labeling system of [IL93] of rank 3. An arrow from a set of vertices to another one indicates that any vertex of the first set is dominated by any vertex of the second. The red arrow indicates a possible pebble move. 95
6.3 An example of the bounded labeling system of [IL93] of rank 3. An arrow from a set of vertices to another one indicates that any vertex of the first set is dominated by any vertex of the second. The red arrow indicates a possible pebble move. 96
6.4 An example of the bounded labeling system of [IL93] of rank 4 in an arbitrary initial configuration (an arrow from a set of vertices to another one indicates that any vertex of the first set is dominated by any vertex of the second). 97

10.1 The three configurations used in the proof of Lemma 10.2 (the numbers represent clock values and the double circles represent crashed vertices). 131
10.2 The three configurations used in the proof of Lemma 10.3 (the numbers represent clock values and the double circles represent crashed vertices). 133
10.3 The three configurations used in the proof of Theorem 10.4 (the numbers represent clock values and the double circles represent crashed vertices). 135
10.4 Example of the execution constructed in case 1 of Theorem 10.4 when $r = 1$ (the numbers represent clock values and the double circles represent crashed vertices). 136
10.5 Initial configuration used in the proof of Theorem 10.5 (the numbers represent clock values and the double circles represent crashed vertex). 137
10.6 The initial configuration for the proof of Theorem 10.6 (the numbers represent clock values and the double circles represent crashed vertices). 139

11.1 An example operation sequence of $\mathcal{SSU}$ on a chain with no faults. Numbers represent clock values. Squared vertex has an enabled rule to be executed. 144
11.2 An example operation sequence of $\mathcal{SSU}$ on a chain with a Byzantine vertex. Numbers are vertex clock values. The Byzantine vertex is in double circle. Squared vertex has an enabled rule to be executed. . . 144
11.3 An example operation sequence of $\mathcal{SSU}$ on a ring with no faults. Numbers represent clock values. Squared vertex has an enabled rule to be executed. 145
11.4 An example operation sequence of $\mathcal{SSU}$ on a chain with a Byzantine vertex. Numbers are vertex clock values. The Byzantine vertex is in double circle. Squared vertex has an enabled rule to be executed. . . 145
11.5 The transitions of in-unison neighbor vertices l and v. An illustration for the proof of Lemma 11.2. 146

11.6 Configuration used in proof of Lemma 11.9 (clock values appear inside vertices and the double circles represent Byzantine vertex). 150
11.7 Configuration used in proof of Lemma 11.10 (clock values appear inside vertices and the double circles represent Byzantine vertex). . . 151
11.8 Configuration used in proof of Lemma 11.11 (clock values appear inside vertices and the double circles represent Byzantine vertex). . . 152
11.9 Configurations used in proof of Proposition 11.4 (clock values appear over vertices and the double circles represent Byzantine vertex). . . . 153
11.10 Configurations used in proof of Proposition 11.5 (clock values appear over vertices and the double circles represent Byzantine vertex). . . . 154

13.1 Summary of respective constraints on Byzantine containment schemes in self-stabilization. An arrow from a scheme to another means that the first is more constrained than the second. 178

14.1 A 0-legitimate configuration for $spec_{NCT}$ (numbers denote the *dist* variable of vertices while the arrow attached to each vertex points the neighbor designated as its parent by the variable *prnt*). Note that r is the (real) root and b is a Byzantine vertex which acts as a (fake) root. 181
14.2 Example of containment areas for BFS spanning tree construction (vertices b_1 and b_2 are Byzantine and vertex r is the root). 186
14.3 Configurations used in proof of Theorem 14.2. 188

15.1 Examples of containment areas for $\mathcal{SP}$. 200
15.2 Examples of containment areas for $\mathcal{F}$. 201
15.3 Examples of containment areas for $\mathcal{R}$. 202
15.4 Configurations used in proof of Theorem 15.3. 216
15.5 Configurations used in proof of Theorem 15.4. 219
15.6 Example of containment areas for *MET*. 220
15.7 Illustration of configurations used in proof of Lemma 15.15, case 1 for the metric *MET* with $c = 1$. 221
15.8 Configurations used in proof of Lemma 15.15, cases 2 and 3. 223

16.1 Summary of respective constraints on Byzantine containment schemes in self-stabilization. An arrow from a scheme to another means that the first is more constrained than the second. 228

17.1 Summary of respective constraints on fault-tolerance schemes used in this thesis. An arrow from a scheme to another means that the first is more constrained than the second. Note that we remove all arrows deductible from transitivity. 237

List of Tables

4.1 Comparison of fault tolerance schemes presented in Chapter 4 67

10.1 Summary of impossibility results 140

12.1 Summary of impossibility results of Chapter 10 157

15.1 Summary of results of Chapter 15 related to $spec_{IMMT}$ with $\mathcal{C}(\mathcal{M}, c)$ a predicate that is true if and only if $\mathcal{M} = (M, W, mr, met, \prec)$ is a strongly maximizable metric and $c \geq max\{0, |M(g)| - 2\}$. 225

16.1 Summary of results of Chapter 15 related to $spec_{IMMT}$ with $\mathcal{C}(\mathcal{M}, c)$ a predicate that is true if and only if $\mathcal{M} = (M, W, mr, met, \prec)$ is a strongly maximizable metric and $c \geq max\{0, |M(g)| - 2\}$. 229

List of Protocols

6.1 Send functions used by our data-link protocol. 87

6.2 Receive functions used by our data-link protocol. 87

6.3 $\mathcal{SDL}$, a self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels with a $(0, 1, 1, 1)$-message performance 89

6.4 Next: the labeling protocol of our stabilizing bounded labeling system. 98

7.1 $\mathcal{PSARS}$: FTPS single-writer multi-reader atomic register simulation (read operation for any vertex v_i, write operation for the writer $w = v_0$ only). 104

7.2 $\mathcal{PSARS}$: Auxiliary functions (for any vertex v_i). 105

11.1 $\mathcal{SSU}$: Minimal and priority $(1, 0)$-strictly stabilizing distributed protocol for $spec_{AU}$ on chains and rings for vertex v. 143

14.1 $\mathcal{SST}$: $(t, 0, n{-}1)$-strongly stabilizing distributed protocol for $spec_{NCT}$ for vertex v. 182

14.2 $\mathcal{SSBFS}$: $(S_B, n-1)$-TA strictly and $(t, S_B^*, n-1)$-TA strongly stabilizing distributed protocol for $spec_{BFST}$ for vertex v. 189

15.1 $\mathcal{SSMMT}$: $(S_B, n-1)$-TA strictly and $(t, S_B^*, n-1)$-TA strongly stabilizing distributed protocol for $spec_{IMMT}$ for vertex v. 203

Chapter 1

Introduction

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

Contents

1.1 Context of the Thesis 2
1.2 Thesis Contributions 4

A long time after his victory against the Lord of the Rings Sauron [Tol37, Tol54a, Tol54b, Tol55], Frodo Baggins got lost in a large labyrinth deep in the Shire. When Samwise Gamgee realized his disappearance, he requested all Hobbits to help him save Frodo. In turn, they entered the labyrinth and spread as evenly as possible. The Hobbits were sufficiently many to cover the whole labyrinth (there was at least one Hobbit per intersection). Each Hobbit was only able to speak with Hobbits that were located at neighboring intersections due to the background noise in the labyrinth. Their goal was to indicate the path of the exit to Frodo by telling him the way he should go at each intersection. However, as it is well known Hobbits love beer and cider, they were all drunk when they entered the labyrinth and after reaching their positions they all fell asleep. When they woke up a few hours later, they were disoriented and each of them indicated an arbitrary neighboring direction to Frodo. Now, even if Hobbits were disoriented, they still wanted to help Frodo. To this purpose, they collaborated to recover from their disoriented state. The Hobbit that was located at the exit of the labyrinth claimed his distinguished status. Hobbits that were located on neighboring intersections heard about it, choose to indicate his way for locating the exit, and propagated the information. Should all Hobbits have done so, they could eventually have indicated a correct path to the exit to Frodo independently of Frodo's initial location in the labyrinth.

Imagine now that some of the very Hobbits that entered the labyrinth were in fact traitors and wanted to get Sauron a revenge by forever loosing Frodo in the labyrinth. When "honest" Hobbits wake up and are disoriented, these "traitor" Hobbits may lie to them in order to reach their goal. These liars do not longer have the same goal than other Hobbits: they want to prevent Frodo to reach the exit while "honest" Hobbits want him to exit. Liar Hobbits may claim that they are located at the exit of the labyrinth even if it is not the case. Then, "honest" Hobbits

propagate this information as previously (they have no mean to decide whether the claim is true). Then, "traitor" Hobbits can attract Frodo to them if he is initially nearer from a liar than from the real exit of the labyrinth.

Building on this intuitive illustration, the core goal of this thesis is to study the impact of the combination of the disorientation of "honest" Hobbits and of lies of "traitor" ones with respect to the ability of "honest" Hobbits to achieve their objective (in our example, indicating a correct path to the exit to Frodo). The sequel of this Introduction is devoted to explain the analogy between this illustrative example and the context of this thesis (Section 1.1) and to the description of contributions of this thesis (Section 1.2).

1.1 Context of the Thesis

Distributed systems and fault-tolerance Distributed computing is a branch of computer science that studies distributed systems. Intuitively, a distributed system is a system constituted from autonomous computational units (called processors, processes, or vertices) enhanced with some communication abilities. Each process can communicate with a subset of other processes. Hence, it is natural to represent communication possibilities of such a system by a graph. Main characteristics of this kind of system are the locality of information (each process have only a local view on the system and must communicate with other processes in order to obtain any global information) and the locality of time, *a.k.a.* asynchronism (each process executes its instructions at its own pace). This model is sufficiently general to capture the main characteristics of any kind of network (LAN, sensor network, peer-to-peer system, ...). In our example, the Hobbits disseminated in the labyrinth may be seen as a distributed system. Indeed, we can represent the labyrinth by a graph (each vertex represents an intersection point) and Hobbits can be viewed as computational units enable to communicate with those located at nearest intersections.

The goal of distributed computing is the design of protocols that solve some problems in distributed systems. Such protocols are called distributed protocols. Generally, the difficulty comes from the fact that considered problems are about achieving global properties of the system. Hence, processes cannot solve it locally and must communicate and cooperate to reach a satisfying global state. In our example, Hobbits in the labyrinth have a global problem to solve: finding a path for Frodo to exit. As no Hobbit has a solution locally, they must cooperate to find such a path. In distributed computing, this problem is well know as the spanning tree construction. A particular process is distinguished as the root of the system (the exit of the labyrinth) and each process must choose as its parent one of its neighbors that is the first process of the path between it and the root (this pointer is denoted in our example by the way each Hobbit indicates to Frodo).

When the size of a distributed system increases or when it is deployed in a dangerous environment, we cannot neglect the probability that some components of the system misbehave (for instance, communications may interfere, some processes

may stop to execute instructions, be subject to attacks or to viruses, ...). Any abnormal comportment of any component of a distributed system is captured by the concept of fault. As the concept of fault is very general, it is classically admitted to classify faults according to several criteria. For instance, duration of the fault can be considered. We distinguish transient faults (that is, faults of finite duration), permanent faults (that is, faults of infinite duration), or intermittent faults (that is, affected processes exhibit successively correct and faulty behavior). We can also distinguish faults by their nature. For example, we can consider crash faults (that is, affected processes simply stop to execute instruction), Byzantine faults (that is, affected processes exhibit an arbitrary behavior), or memory corruption. In our example, the drunkenness of Hobbits can be seen as a transient fault (Hobbits wake up disoriented in a finite time) that induces a memory corruption (Hobbits are disoriented when they wake up) while the betrayal of "traitor" Hobbits can be seen as a permanent Byzantine fault (these Hobbits do not longer cooperate with others to achieve a global task).

Distributed fault-tolerance is the sub-branch of distributed computing that focuses on distributed protocols that are resilient to faults. That is, a fault-tolerant distributed protocol ensures that some properties are satisfied even if faults strike the system. There exists several notions in fault-tolerance depending on the tolerated classes of faults. In the following, we present the most important fault-tolerance schemes studied in this thesis.

Self-stabilization In our illustrative example, when all Hobbits are honest, they eventually succeed to indicate Frodo a (shortest) path to the exit even if they are disoriented after waking up. This ability to recover a correct behavior in a finite time from any arbitrary initial state is called self-stabilization in distributed computing. Self-stabilization allows transient fault tolerance (whatever the nature of this fault is). Indeed, when a transient fault ends, the state of the system may be arbitrary (due to abnormal actions during the fault, memory corruption, ...). Then, a self-stabilizing distributed protocol ensures that it recovers a correct behavior in a finite time without any external or manual help. However, a self-stabilizing distributed protocol can only deal with transient faults. Indeed, such a distributed protocol relies on the assumption that the protocol of each process is not corrupted during the transient fault (only the volatile memory may be affected by transient faults) and that each process executes correctly its protocol after the end of transient faults. Otherwise, the distributed protocol may never return to a desirable behavior.

Containment of crash/Byzantine faults in self-stabilization To ensure better fault resilience, it is possible to consider self-stabilizing distributed protocols that are moreover able to deal with a (limited) number of permanent or intermittent faults after the end of transient faults. Due to this challenging model of distributed systems, we cannot ensure that the whole distributed system retrieves a correct behavior in a finite time as in self-stabilization (at least, some faulty processes may

exhibit abnormal behavior infinitely). In our example, when there are some "traitor" Hobbits that want to loose Frodo, the protocol used by honest Hobbits only ensures that Frodo finds the exit if he is initially (strictly) nearer from the exit than from a liar Hobbit. This fault-tolerance property could be seem weak at the first glance but we prove in this thesis that we cannot ensure a stronger one if honest Hobbits are initially disoriented.

The main problem addressed in this thesis is joint tolerance to transient faults and to permanent or intermittent faults in distributed systems. The main difficulty comes from the following fact: in such systems, it is impossible to distinguish a (permanently or intermittently) faulty process (that does not cooperate to reach the global goal of the distributed system) from a badly initialized but honest process (that still cooperate to reach the common goal in spite of transient faults). In our illustration, "traitor" Hobbits may attract Frodo to the wrong direction because honest Hobbits cannot distinguish the honest Hobbit at the exit of the labyrinth from a liar and thus propagate the information in a similar way in both cases.

1.2 Thesis Contributions

This section summarizes this thesis' contributions in the context of joint tolerance to transient faults and to permanent or intermittent faults in distributed systems. This thesis is organized in four parts. The first one presents the context of the thesis while the three others focus on three independent fundamental problems in distributed systems: the simulation of a strong computational model over a weaker one, clock synchronization, and spanning tree construction.

Part One: Context The first part of this thesis is devoted to the formal definition of a distributed system and to a survey on fault-tolerance in such systems. First, Chapter 2 provides a detailed description of distributed systems, a formal presentation of computational models used in this thesis, and a model for faults. Then, Chapter 3 focuses on a particular notion of our model, called daemon, that gather assumptions related to scheduling. We provide a general taxonomy that allows the comparison of every daemon already defined in self-stabilization. Finally, Chapter 4 surveys fault-tolerance in distributed systems focusing mainly on variants of self-stabilization that tolerate moreover permanent or intermittent faults.

Part Two: Atomic Register The second part of this thesis extends results presented in [AAD+10, DDPBT11a, AAD+11, DDPBT11b]. We focus on a computational model transformer. As a matter of fact, there exists several computational models in distributed systems (that are mainly characterized by their atomicity level, *i.e.*, by actions that are assumed to be executed atomically). The higher the atomicity model, the simpler the design of a distributed protocol, the lower its realism. Computational model transformers allow the design of distributed protocols in a high atomicity computational model while permitting to execute them in low

atomicity ones. Chapter 7 presents such a computational model transformer for distributed systems subject to transient and permanent crash faults. This transformer makes use of two independent tools that are presented in Chapter 6.

Part Three: Unison The third part of this thesis is related to results from [DPBT09, DPBNT10, DPBT11]. We focus on clock synchronization in distributed systems that are simultaneously subject to transient and intermittent Byzantine faults. This challenging environment obviously prevents perfect synchronization for every process. Hence, we concentrate on a weaker synchronization problem that only requires that clocks of neighboring processes are "close" from each other. Even with this weaker problem, Chapter 10 lists numerous impossibility results related to the number of faults, the scheduling, and the topology of the distributed system. We provide in Chapter 11 an optimal distributed protocol for the remaining possible cases.

Part Four: Spanning Tree The fourth part of this thesis summarizes results presented in [DMT10c, DMT10a, DMT10b, DMT11a, DMT11c, DMT11b]. The main goal of this part is the construction of spanning trees in presence of both transient and intermittent Byzantine faults. Spanning tree construction consists in the construction of a communication subgraph that spans the whole system with a minimal number of communication links. To this purpose, we introduce three new concepts of joint tolerance to transient and intermittent Byzantine fault tolerance. These concepts are characterized by the fact that the effects of Byzantine faults are contained in some portions of the distributed system. In Chapter 14, we present distributed protocols for two particular cases of spanning trees while Chapter 15 deals with a large class of spanning trees. We prove that all distributed protocols presented in this part achieve the best possible Byzantine containment.

Reading map Even if the three last parts of this thesis are independent, there exists some dependencies between chapters of this thesis. Figure 1.1 summarizes them.

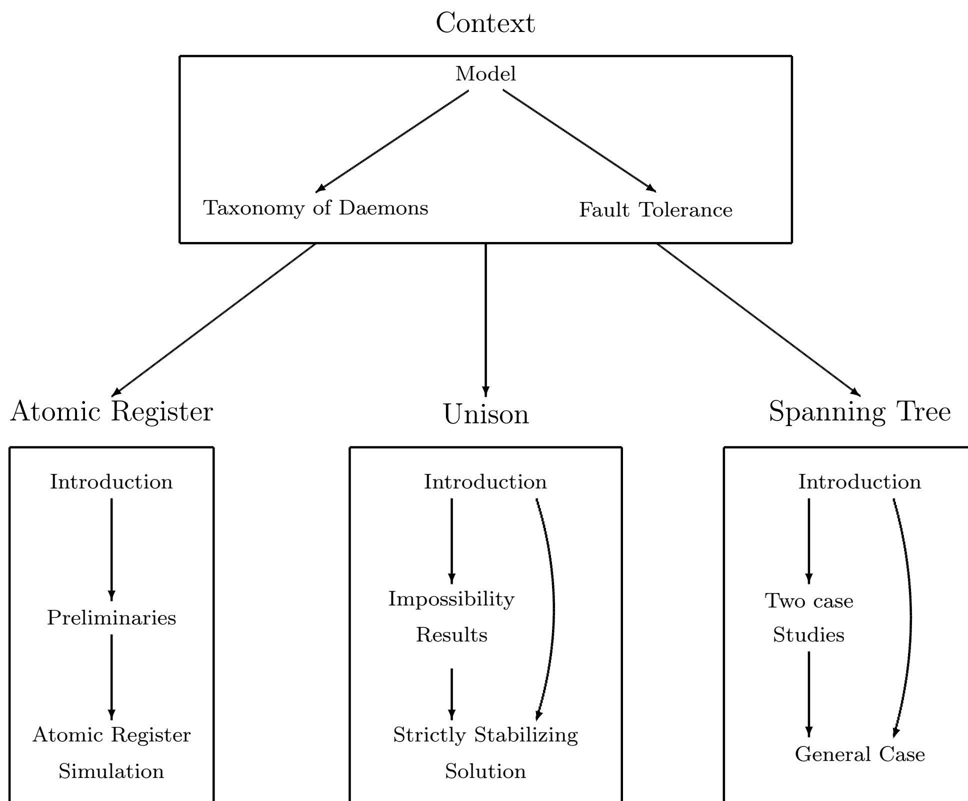

Figure 1.1: Dependencies between chapters of this thesis.

Part I

Context

Chapter 2
Model

> Essentially, all models are wrong, but some are useful.
>
> George E. P. Box

Contents

2.1 Distributed System 9
 2.1.1 Characteristics 10
 2.1.2 Advantages 10
2.2 Communication Graph 11
2.3 Models Used in this Thesis 13
 2.3.1 Generic Model 13
 2.3.2 State Model 15
 2.3.3 Message Passing Model 16
2.4 Fault Taxonomy 18
 2.4.1 Faults 18
 2.4.2 Fault Patterns 21

In this chapter, we first present generalities about distributed systems (Section 2.1) and we provide then the model used in this thesis. This model is composed of the following elements:

- a communication graph that represents the communication capacities of the distributed system (see Section 2.2),
- a computational model that gathers assumptions on asynchronism, atomicity of executions, etc. of the distributed system (see Section 2.3), and
- a fault model that allows to describe any abnormal behavior in the distributed system (see Section 2.4).

This chapter introduces numerous definitions and notations that are extensively used in this thesis. For convenience, the reader can find at the end of this thesis an index and a list of notations that summarize those that are used in the remainder of this thesis.

2.1 Distributed System

In computer science, a distributed system (see *e.g.* [Tel10, Lyn96]) is a set of computation unit (potentially disseminated geographically) enhanced with communication capacities. These computation units, usually called processes, cooperate

together in order to solve a given task. This cooperation includes intern computations and information exchanges. Information is exchanged using communication links that connect processes.

Note that this definition is enough general to enclose systems such that computer networks, sensor networks, parallel computers, or peer-to-peer systems. Actually, such systems share some common properties such as information and time locality. In this section, we first describe characteristics of distributed systems (with respect to centralized systems, that is system containing only one process) and we present main advantages to use such systems.

2.1.1 Characteristics

As distributed systems are potentially composed of a large number of heterogeneous elements, their main characteristic is locality. We can mainly distinguish two localities: the information locality and the time locality.

Information locality Each process has access only to its local information at a given time. If it needs supplementary information, it must communicate with its neighbors (that is, processes that share a communication link with it). In other words, no process has a global knowledge on the system.

Time locality As elements of a distributed system may be heterogeneous, their clocks are not synchronized and may deviate from each others at any speed during the system life time. Moreover, communication speed may be very different between two pairs of neighbors. In other words, distributed systems are fully asynchronous.

These characteristics are the main challenge to overcome when designing distributed protocols (that is, description of actions that processes must execute) that solve global problems on distributed systems.

2.1.2 Advantages

Whereas characteristics of a distributed system lead to larger difficulties in developing protocols as in a centralized system, the former provides some advantages that we present in the following.

Information exchange The first motivation of a distributed system is to allow data exchange at large scale. For example, the Internet allows nowadays millions of people to communicate (via e-mail, chatting, *etc.*) and to share huge amount of data.

Resource sharing A distributed system allows its users to access to any resource (memory, computation power, hard disk space, *etc.*) disseminated in the whole distributed system. In this way, each user of the distributed system can access to a greater amount of resources than with a centralized system. On the other hand, that implies that we must deal with problems due to the concurrent access to resources.

Increasing the computation power Peer-to-peer systems (P2P, see *e.g.* [Ora01, LMSM09]) are a particular class of distributed systems that allows to share its computation power between its users. In this way, distributed systems allows concurrent computations that lead to a reduction of computation time (if the computation can be parallelized).

For example, the project SETI@HOME of the University of Berkeley uses the computation power of over 3 millions of computers connected to the Internet in order to detect intelligent life outside Earth by analyzing radio signals from space.

Fault tolerance A fault is a temporary or permanent failure of a component of the distributed system. With contrast to a centralized system, a part of services may continue to work correctly after a fault in a distributed system.

On the other hand, note that the probability of a failure increases with the number of components of the system. Note also that, in case of failures, the validity of the results or of the behavior of the whole distributed system may be not guaranteed. We describe in Chapter 4 main techniques to ensure fault tolerance in distributed systems.

2.2 Communication Graph

As we previously discussed, a distributed system consists of a set of processes that have some communication capacities. The most natural way to model these communication capacities is to use a graph. The processes are vertices in this graph (V denotes the set of vertices) and the edges of this graph are pairs of processes that can communicate with each other (E denotes the set of edges with $E \subseteq V^2$). Such pairs of vertices are called neighbors. Hence, $g = (V, E)$ forms the communication graph of the distributed system. In this thesis, we use some graph theory notions to describe properties of distributed systems (for more informations about graph theory, see [Ber67]).

We will consider only non oriented and simple communication graph, that is, for any edge $\{u, v\}$ of E, the vertex u is able to communicate with v and the vertex v is able to communicate with u (we also talk of bi-directional communication) and there exists no loop in g (that is, edges of the form $\{v, v\}$ are forbidden).

Communication graphs considered in this thesis may be either identified or anonymous. In the first case, each vertex of the communication graph has a distinct identity whereas in the second case, vertices are indistinguishable from each other.

In the following, we present some useful definitions and notations.

A path $(v_1, v_2, \dots, v_k)$ of g is a sequence of vertices of V such that, for any $i \in \{1, \dots, k-1\}$, we have $\{v_i, v_{i+1}\} \in E$. The length of a path $(v_1, v_2, \dots, v_k)$ of g is the number of edges of this path (*i.e.* $k-1$). A path $(v_1, v_2, \dots, v_k)$ of g is elementary if, for any $i \in \{1, \dots, k\}$ and any $j \in \{1, \dots, k\} \setminus \{i\}$, we have $v_i \neq v_j$. A cycle $(v_1, v_2, \dots, v_k)$ of g is a path of g such that $v_1 = v_k$. A cycle $(v_1, v_2, \dots, v_k)$ of g is elementary if $(v_1, v_2, \dots, v_{k-1})$ is an elementary path.

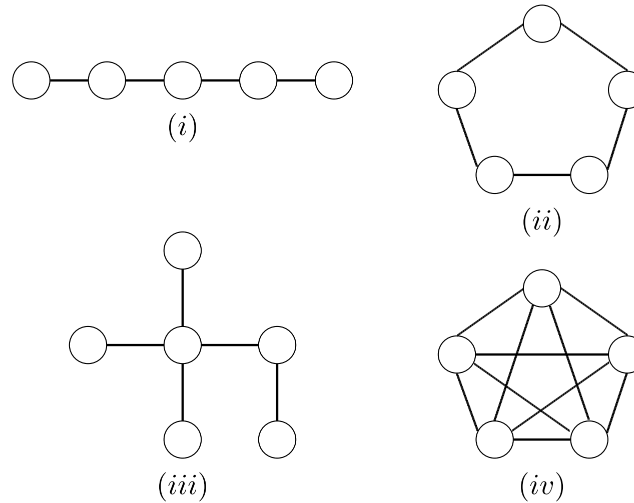

Figure 2.1: Some classical communication graphs: (i) a chain, (ii) a ring, (iii) a tree, and (iv) a complete communication graph.

A communication graph g is connected if, for any pair of disjoints vertices u and v, there exists a path $(v_1, v_2, \dots, v_k)$ of g such that $v_1 = u$ and $v_k = v$.

We can now introduce some classical communication graph topologies used in this thesis. A communication graph is a chain when it is reduced to an elementary path. A communication graph is a ring when it is reduced to an elementary cycle. A communication graph is a tree when it is a connected communication graph and $|E| = |V| - 1$. A communication graph is complete when $E = V^2 \setminus \{\{v, v\} | v \in V\}$. Some examples of such topologies are provided by Figure 2.1.

Given a communication graph $g = (V, E)$, the communication subgraph induced by a subset $V' \subseteq V$ is the communication graph $g' = (V', E')$ where $E' = \{\{u, v\} \in E | u \in V' \wedge v \in V'\}$.

Given a communication graph $g = (V, E)$, a spanning tree of g is a communication subgraph induced by V that is a tree and a spanning forest of g is a communication subgraph induced by V such that each of its connected components is a tree.

Given a communication graph g, we introduce the following set of notations. Firstly, n denotes the number of vertices of the graph whereas m denotes the number of edges ($n = |V|$ and $m = |E|$). For any vertex v, N_v denotes the set of neighbors of v. The distance between two vertices u and v (that is, the length of a shortest path between u and v in g) is denoted by $dist(g, u, v)$. The diameter of g (that is, the maximal distance between two vertices of g) is denoted by $diam(g)$. For any vertex v, the degree of v (that is, the number of neighbors of v) is denoted by $deg(g, v)$. The maximal degree of g (that is, the maximal number of neighbors of a vertex in g) is denoted by $deg(g)$.

2.3 Models Used in this Thesis

It is usually admitted to describe the distributed system communication capacities by a communication graph (see Section 2.2). However, there exists several computational models for distributed systems. These models are distinguished by their atomicity level. Atomicity depends on actions that are allowed in a single atomic step. A few examples of classical computational models follow (from higher to lower atomicity level):

1. State model (or shared memory model) [Dij74]: this is the classic model for self-stabilization area (see Section 4.1) for which in one atomic step, a vertex can read the state of all its neighbors, and update its own state;
2. Shared register model [DIM93]: in one atomic step, a vertex can read the state of one of its neighbors, or update its own state, but not both simultaneously;
3. Message passing model [Tel10]: this is the classical model for distributed systems for which in one atomic step, a vertex sends a message to one of its neighbors, or receives a message from one of its neighbors, but not both simultaneously.

In this thesis, we use only the message passing model in Part II and the state model in Parts III and IV. We formally describe them in the following of this section. First, we provide a generic model for distributed system in Section 2.3.1 that needs some supplementary definitions to describe both models (see respectively Sections 2.3.2 and 2.3.3).

2.3.1 Generic Model

This section presents a generic computational model for distributed systems. Once this generic computational model defined, we can instantiate an actual computational model by providing a few *ad-hoc* definitions.

Distributed Protocol Given the communication graph that describes the communication capacities of the distributed system, the state of this communication graph (values of variables, registers, communication links...) at a given time is a configuration. The actual definition of a configuration depends on the chosen computational model. Each variable, register, communication link, *etc.* of the system is called a member. Each member of the system is attached to several vertices that have the ability to modify the value of this member. We denote by M the set of members of the whole system. The set of configurations of g is denoted by Γ. For any configuration $\gamma \in \Gamma$ and any member $\mu \in M$, $\gamma(\mu)$ denotes the value of μ in γ.

An action α of g transitions the graph from one configuration to another. The set of actions of g is denoted by $\tilde{A}$ ($\tilde{A} = \Gamma^2$). Depending on the chosen computational model, there exists some constraints on actions allowed by the computational model (maximal number of modifications of variables by a vertex, operations on registers or communication links...). We denote the set of allowed actions by the computational

model by A. A constraint shared by any computational model is the following: any action modifies the configuration of the system. In other words, $A \subseteq \tilde{A} \setminus \{(\gamma, \gamma) | \gamma \in \Gamma\}$.

A distributed protocol π on g is defined as a subset of A that gathers all actions of g allowed by π. The set of distributed protocols on g is denoted by Π ($\Pi = P(A)$ [1]).

Execution Given a communication graph g, a distributed protocol π on g, an execution σ of π on g starting from a given configuration γ_0 is a maximal sequence of actions of π of the following form $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)(\gamma_2, \gamma_3)\ldots$. An execution is maximal if it is either infinite or finite but, in this last case, the last configuration of the execution is terminal (that is, there exists no actions of π starting from this configuration). The set of all executions of π on g starting from all configurations of Γ is denoted by Σ_π. The set of all executions of all distributed protocols on g starting from all configurations of Γ is denoted by Σ_Π ($\Sigma_\Pi = \{\Sigma_\pi | \pi \in \Pi\}$).

Daemon The asynchronism of executions is captured by an abstraction called daemon (this abstraction was introduced by [Dij74]). Intuitively, a daemon restricts executions of distributed protocols to a set of executions that share some given properties. Formal definition follows.

Definition 2.1

Given a communication graph g, a daemon d on g is a function that associates to each distributed protocol π on g a subset of executions of π.

$$\begin{array}{cccll} d & : & \Pi & \longrightarrow & P(\Sigma_\Pi) \\ & & \pi & \longmapsto & d(\pi) \in P(\Sigma_\pi) \end{array}$$

The set of all daemons on g is denoted by $\mathcal{D}$.

Hence, the daemon allows us to define some restrictions on the scheduling of executions as we consider only a subset of executions as allowed by the daemon. That is, we assume that possible executions of the distributed system are only those defined by the daemon. More formally,

Definition 2.2

Given a communication graph g, a daemon d on g and a distributed protocol π on g, an execution σ of π ($\sigma \in \Sigma_\pi$) is allowed by d if and only if $\sigma \in d(\pi)$.

Definition 2.3

Given a communication graph g, a daemon d on g and a distributed protocol π on g, we say that π runs on g under d if we consider that only possible executions of π on g are those allowed by d.

1. where, for any set S, $P(S)$ denotes the set of parts of S.

Whereas it is possible to define arbitrary daemons, only few of them are interesting in practice. Interesting properties of daemons and classical daemons are discussed in details in Chapter 3. Two main criteria are usually distinguished:

1. Spatial assumptions: at each action, we can make restrictions on the location of vertices chosen by the daemon (*e.g.* no two neighboring vertices are simultaneously chosen by a locally central daemon).
2. Temporal assumptions: we can restrict choices of the daemon on the whole execution (*e.g.* no vertex can be chosen more than k times between two consecutive choices of any other vertex by a k-bounded daemon).

Other Notations Given a communication graph g and a distributed protocol π on g, we introduce the following set of notations.

Each action of g is characterized by the set of vertices that are activated (or scheduled) during this action, *i.e.* that modify the value of at least one member attached to them. The actual definition of the activation depends on the chosen model. We define the following function:

$$\begin{array}{rcrcl} Act & : & A & \longrightarrow & P(V) \\ & & \alpha & \longmapsto & \{v \in V | v \textit{ is activated during } \alpha\} \end{array}$$

A vertex v is enabled by π in a configuration γ if and only if there exists an action $\alpha = (\gamma, \gamma') \in \pi$ such that v is activated during α.

Each configuration of g is characterized by the set of vertices enabled by π in this configuration. We define the following function:

$$\begin{array}{rcrcl} Ena & : & \Gamma \times \Pi & \longrightarrow & P(V) \\ & & (\gamma, \pi) & \longmapsto & \{v \in V | v \textit{ is enabled by } \pi \textit{ in } \gamma\} \end{array}$$

2.3.2 State Model

This section is devoted to the definition of the state model using our generic model (see Section 2.3.1). Recall that the state model is the classical model for the self-stabilization area and is a high-atomicity model. Indeed, in each action, any vertex can read the state of each of its neighboring vertices and modifies its own state.

Characteristics of the State Model Each vertex of g has a set of variables, each of them ranges over a fixed domain of values. A state $\gamma(v)$ of a vertex v is the vector of values of all variables of v at a given time. An assignment of values to all variables of the graph is a configuration.

In this model, any action is allowed provided that it effectively modifies the configuration of the communication graph, that is $A = \tilde{A} \setminus \{(\gamma, \gamma) | \gamma \in \Gamma\}$.

A vertex v is activated during an action $\alpha = (\gamma, \gamma')$ if and only if $\gamma(v) \neq \gamma'(v)$.

Guarded Representation of Distributed Protocols For the sake of clarity, we cannot describe distributed protocols by enumerating all their actions. Therefore, we choose to present distributed protocols using a local description of actions borrowed from [Dij74]. Each vertex has a local protocol consisting of a set of guarded rules of the following form:

$$< label > :: < guard > \longrightarrow < action >$$

where:

- $< label >$ is a name to refer to the rule in the text.
- $< guard >$ is a predicate that involves variables of the vertex and of its neighbors. This predicate is true if and only if the vertex is enabled in the current configuration. We say that a rule is enabled in a configuration when its guard is evaluated to true in this configuration.
- $< action >$ is a set of instructions modifying the state of the vertex. This set of instructions must describe the changes of the vertex state if this latter is activated by the daemon.

2.3.3 Message Passing Model

In this section, we define the message passing model using our generic model (see Section 2.3.1). The message passing model is a low-atomicity model. Indeed, in each atomic action any vertex can only execute an internal instruction and one communication operation (namely send or receive a message) with respect to only one neighbor.

This model is more realistic than the state model since actual distributed systems exchange information by message passing. There exists several variants of this model according to assumptions on communication channels properties (see *e.g.* [Lyn96]). Main common assumptions follow.

1. the communication channels are either synchronous (their delivery time is bounded) or asynchronous (their delivery time is not bounded);
2. the communication channels are reliable (each sent message is delivered) or not;
3. the capacity of each communication channel (the number of distinct messages that the channel can contain) is either unbounded or bounded; and
4. the communication channels are FIFO (First In-First Out: the delivery order of messages is identical to the sending order) or not.

In this thesis, we consider a message passing model in which:

1. the communication channels are asynchronous (their delivery time cannot be bounded);
2. the communication channels are not reliable but are fair (a message that is infinitely often sent is infinitely often delivered);

3. the capacity of each communication channel is bounded by an integer c (each communication channel may contain up to c messages), we assume that the capacity c is known to the protocol and is fixed over time and that the sending of a message through a full communication channel leads to the loose of an arbitrary message; and
4. the communication channels are non-FIFO. That is, the order of delivery of messages is independent from the order of sending.

In the following of this section, we only provide the description of the message passing model used in this thesis with our generic model (the derivation to any other message passing model is straightforward).

Characteristics of the Message Passing Model Each vertex of g has a set of variables, each of them ranges over a fixed domain of values. Moreover, each vertex has a program counter that indicates the next instruction to execute. A state $\gamma(v)$ of a vertex v is the vector of values of all variables and of the program counter of v at a given time. Each edge $e = \{u, v\}$ of g has 4 c-sets of messages where c is an integer (rather than a queue in order to reflect the non-FIFO property), namely $\overrightarrow{uv}$ (that stores messages sent by u to v), $\overleftarrow{uv}$ (that stores acknowledgments sent by v to u), $\overrightarrow{vu}$ (that stores messages sent by v to u), and $\overleftarrow{vu}$ (that stores acknowledgments sent by u to v). Acknowledgments are a particular class of messages that are used to confirm the reception of a given message to the sender. Each acknowledgment contains a copy of the original message in order to track the message that causes its sending. A state $\gamma(e)$ of an edge e is the vector of values of all m-sets of e at a given time. An assignment of values to all variables of vertices and all c-sets of edges of the graph is a configuration.

An action is allowed by the message passing model if and only if, during this action, each vertex v modifies its state and executes only one of the following communication operation:

send(m,s): (where m is an arbitrary message and $s \in \{\overrightarrow{vu}, \overleftarrow{uv}\}$) that places the message m in the c-set s. That is, s is replaced by $s \cup \{m\}$ (if the obtained union does not respect the bound $|s \cup \{m\}| \leq c$, then an arbitrary message in the obtained union is deleted).

receive(m,s): (where m is a variable of v that can contain an arbitrary message and $s \in \{\overrightarrow{uv}, \overleftarrow{vu}\}$) that removes an arbitrary message from the c-set s and copies it on m (if the c-set s is empty then the function returns $\perp$, the null message).

A vertex v is activated during an action $\alpha = (\gamma, \gamma')$ if and only if $\gamma(v) \neq \gamma'(v)$ or v executes a communication operation during α.

Representation of Distributed Protocols As in the state model, we cannot provide a full description of a distributed protocol by enumerating all actions that compose it. We choose to provide a local description of distributed protocols using a classical pseudo-code.

2.4 Fault Taxonomy

When the number of components in a distributed system increases, the possibility that some of them fail also increases. In other words, the probability of failure in a large-scale distributed system cannot be ignored. In the same way, we cannot ignore failures in distributed systems like sensor networks (since components have limited energy, are deployed in dangerous environments...). Distributed systems may also encounter attacks, viruses, *etc.*. A fault is the consequence of a failure or of any undesirable event on the distributed system. For example, a vertex stops to operate (crash fault) due to a power failure.

As many works interest in fault-tolerance in distributed systems, there exists several models of faults (see *e.g.* [Lyn96, Dol00, Tel10, AW04, GR06, ADGF$^+$07, Tix09, Ray10, Ray00]). However, it is generally admitted to classify faults according to several criteria that we describe in the following.

Nature: Faults are characterized by the kind of perturbation they introduce in the distributed system. For example, a crash fault leads affected components to stop their execution, a Byzantine fault induces an arbitrary (and potentially malicious) behavior while de-sequencing faults arbitrarily re-order messages in a communication channel. Note that there exists many classes of faults for describing any undesirable behavior of a distributed system.

Duration: We can distinguish faults by their duration. A fault can be permanent (of infinite duration), transient (of finite duration), or intermittent (affected components are successively faulty and correct).

Span: This third characteristic focuses on the number of components affected by the fault.

Note that, unlike works on dependability (see *e.g.* [ALRL04]), we do not consider what caused the fault (failure, hacking, bug, cosmic radiation,...) but only its impact on the distributed system behavior (memory corruption, erroneous execution,...).

The aim of this section is to formally define a taxonomy for faults in our generic computational model (see Section 2.3.1) that allows to describe this informal classification. We first provide a formal definition of a fault and a possible classification for them (see Section 2.4.1). Then, we define a fault pattern as a set of faults that describe all possible faulty behaviors in an execution. This is sufficient to formally define all fault models considered in this thesis (see Section 2.4.2).

2.4.1 Faults

In this section, we define formally a fault in our generic model of distributed system (see Section 2.3.1). Then, we provide illustrations of our definition by modeling classical faults as transient faults, permanent faults, Byzantine faults,...

Definition Intuitively, a fault is a portion of execution (finite or not) in which a subset of vertices executes only actions that do not belong to the protocol (in

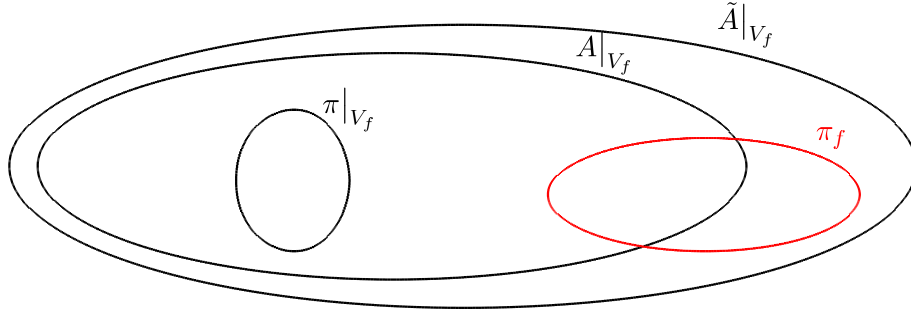

Figure 2.2: Illustration of Definition 2.5.

some cases, these actions may be not allowed by the model as a message lost in the message passing model). For example, a Byzantine fault may allow any actions while a crash fault allows only void actions (that is, actions that do not modify the state of affected components) that are normally not allowed by the computational model. This modeling of faults by actions is borrowed from [KA98]. A fault is hence characterized by its bounds (in time and in space) and by a set of actions that are executed during this fault.

Before stating the formal definition of a fault, we need to introduce a tool called projection. The projection of an action on a communication subgraph induced by $V' \subseteq V$ is the action of this communication subgraph that modifies only members of the communication subgraph in the same way as the original action.

Definition 2.4 (*Projection*)

Given a communication graph g, a configuration $\gamma \in \Gamma$ of g, an action $\alpha = (\gamma, \gamma') \in \tilde{A}$ of g, a set of actions $S \in P(\tilde{A})$, an execution $\sigma = \alpha_0\alpha_1\dots$ of a distributed protocol π on g, and a subset $V' \subseteq V$ that induces the communication subgraph g' of g,

- *the projection of γ on V' denoted by $\gamma|_{V'}$ is the configuration of g' such that for any $\mu \in M'$, we have $\gamma|_{V'}(\mu) = \gamma(\mu)$ where M' denotes the set of members of g'.*
- *the projection of α on V' denoted by $\alpha|_{V'}$ is the action of g' defined by $\alpha|_{V'} = (\gamma|_{V'}, \gamma'|_{V'})$.*
- *the projection of S on V' denoted by $S|_{V'}$ is the set of actions of g' defined by $S|_{V'} = \{\alpha|_{V'} | \alpha \in S\}$.*
- *the projection of σ on V' denoted by $\sigma|_{V'}$ is the execution of g' defined by $\sigma|_{V'} = \alpha_0|_{V'}\alpha_1|_{V'}\dots$.*

Definition 2.5 (*Fault*)

Given a communication graph g and a distributed protocol π on g, a fault f is a quadruplet (V_f, b_f, e_f, π_f) where $V_f \subseteq V$ is the span of f, $b_f \in \mathbb{N}$ is the beginning point of f, $e_f \in \{b_f + 1, b_f + 2, \dots\} \cup \{\infty\}$ is the end point of f, and $\pi_f \in P((\tilde{A} \setminus \pi)|_{V_f})$ is the behavior of f.

Figure 2.2 illustrates the definition of the behavior of a fault: π_f is a subset of actions of the communication subgraph induced by V_f that has no intersection with the projection of the distributed protocol. Note that some actions not allowed by the computational model may belong to π_f.

We say that a fault $f = (V_f, b_f, e_f, \pi_f)$ affects an execution $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots$ (or that the execution σ is subject to the fault f) if vertices of V_f exhibit the behavior π_f in the portion of execution delimited by γ_{b_f} and γ_{e_f} (that is, vertices of V_f execute actions of π_f only during this portion of execution while other vertices executes always actions of the protocol).

Definition 2.6 (*Execution subject to a fault*)

Given a communication graph g, an execution $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots$ of a distributed protocol π is subject to a fault $f = (V_f, b_f, e_f, \pi_f)$ if:

$$\forall i \in \mathbb{N}, \begin{cases} i < b_f \Rightarrow (\gamma_i, \gamma_{i+1}) \in \pi \\ b_f \leq i < e_f \wedge (\exists v \notin V_f, v \in Act(\gamma_i, \gamma_{i+1})) \Rightarrow (\gamma_i, \gamma_{i+1})\big|_{V \setminus V_f} \in \pi\big|_{V \setminus V_f} \\ b_f \leq i < e_f \wedge (\exists v \in V_f, v \in Act(\gamma_i, \gamma_{i+1})) \Rightarrow (\gamma_i, \gamma_{i+1})\big|_{V_f} \in \pi_f \\ i \geq e_f \Rightarrow (\gamma_i, \gamma_{i+1}) \in \pi \end{cases}$$

Modeling of classical faults As we already stated, there exists some classical criteria to distinguish faults from each other. In the following, we describe them using our fault model. The first one is localization in time. Usually, two kinds of faults are distinguished. Transient faults are faults of finite duration whereas permanent faults have infinite duration. With our model of faults, given a fault $f = (V_f, b_f, e_f, \pi_f)$, we have:

- f is transient if and only if $e_f \neq \infty$.
- f is permanent if and only if $e_f = \infty$.

Secondly, we can characterize faults by their span, that is the number and the location of vertices that are affected. For example, a fault is global if the whole system is affected and is local if only one vertex is affected. With our model of faults, given a fault $f = (V_f, b_f, e_f, \pi_f)$, we have:

- f is local if and only if $|V_f| = 1$.
- f is global if and only if $V_f = V$.

A third criterion is the nature of the fault, that is the behavior of faulty vertices. Two classical examples are crash faults in which affected vertices simply stops to execute actions and Byzantine faults in which affected vertices can exhibit an arbitrary behavior (this kind of faults can obviously cause the most harm). With our model of faults, given a fault $f = (V_f, b_f, e_f, \pi_f)$, we have:

- f is a crash fault if $\pi_f = \{(\gamma, \gamma)\big|_{V_f} | \gamma \in \Gamma\}$.
- f is a Byzantine fault if $\pi_f = (\tilde{A} \setminus \pi)\big|_{V_f}$.

Other examples This model of faults allows us to describe any fault. For example, in the message passing model, an omission fault (that is, a fault in which a vertex "forgot" to execute some action) can be described by a set of actions that corrupt the program counter without modifying other variables. Messages looses are described by a set of actions that delete messages in channels whereas no vertex executes the receive communication operation.

2.4.2 Fault Patterns

In the previous section, we define the notion of fault but we do not consider executions that may be subject to several faults (see Section 2.4.1). This is the goal of this section. We first define a fault pattern as a set of faults that affects simultaneously an execution and we provide examples of classical fault patterns.

Definition A fault pattern is simply a set of faults. Note that we add no constraints on this definition in order to keep the possibility to describe any set of faults by a fault pattern.

Definition 2.7 (*Fault pattern*)

Given a communication graph g, a fault pattern FP is a set of fault $FP = \{f_1, f_2, \ldots\}$.

Intuitively, an execution is subject to a fault pattern if this execution is simultaneously subject to all faults of the fault pattern. In order to define formally this notion, we must first define the behavior of a system simultaneously subject to several faults. We provide an operator to describe this behavior given the behavior of each fault of the fault pattern.

Given two actions $\alpha_1 = (\gamma, \gamma_1)$ and $\alpha_2 = (\gamma, \gamma_2)$, we define a merged action of α_1 and α_2 as the action α_1 itself, the action α_2 itself, or an action corresponding to the simultaneous execution of α_1 and α_2. Note that, in the last case, this merged action is not unique since it may exists some conflicts between α_1 and α_2 (that is, both actions may modify the same member in two different ways). In this case, the value of each member of the resulting configuration of the merged action:

- is equal to its value in γ if neither α_1 nor α_2 modify this member,
- is equal to its value in γ_1 if only α_1 modifies this member,
- is equal to its value in γ_2 if only α_2 modifies this member, and
- is equal to its value in γ_1 or in γ_2 if both actions modifies this member.

Finally, given two actions $\alpha_1 = (\gamma_1, \gamma_1')$ and $\alpha_2 = (\gamma_2, \gamma_2')$ with $\gamma_1 \neq \gamma_2$, we define a merged action of α_1 and α_2 as the action α_1 itself or the action α_2 itself only.

The formal statement of this definition follows.

Definition 2.8 (*Merged actions*)

Given a communication graph g and two actions of g $\alpha_1 = (\gamma_1, \gamma_1') \in \tilde{A}$ and $\alpha_2 = (\gamma_2, \gamma_2') \in \tilde{A}$, the set of merged actions of α_1 and α_2 denoted by $\alpha_1 \otimes \alpha_2$ is

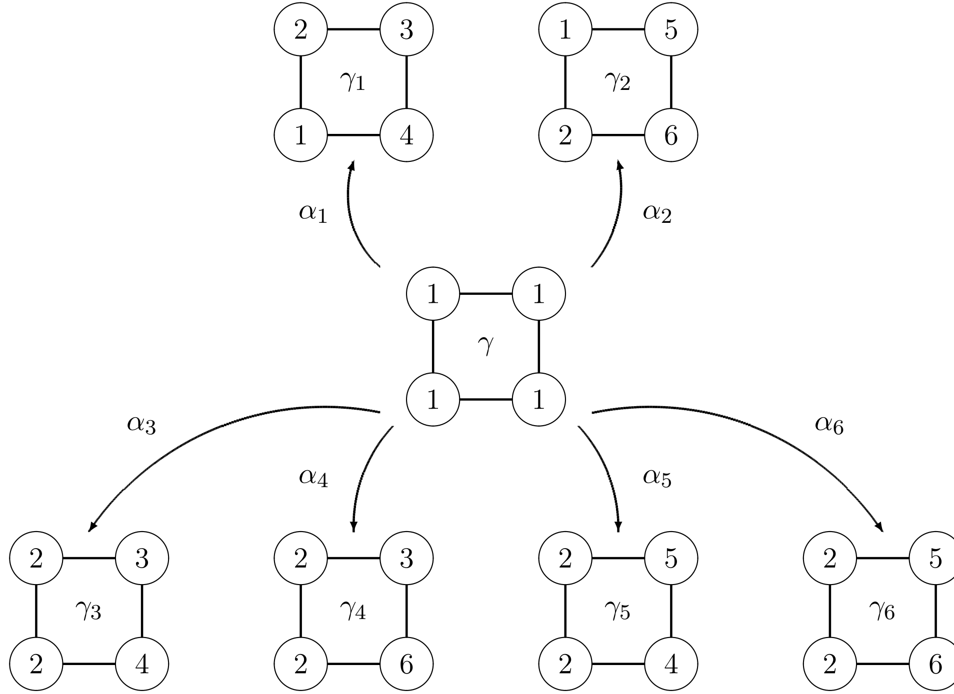

Figure 2.3: Given two actions α_1 and α_2, we have $\alpha_1 \otimes \alpha_2 = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}$ (each number represents the value of a member of the system).

defined by:

$$\alpha_1 \otimes \alpha_2 = \begin{cases} \{\alpha_1, \alpha_2\} & \text{if } \gamma_1 \neq \gamma_2 \\ \{\alpha_1, \alpha_2\} \bigcup \\ \quad \{\alpha = (\gamma, \gamma') | \forall \mu \in M, \\ \quad (\gamma'_1(\mu) = \gamma(\mu) \wedge \gamma'_2(\mu) = \gamma(\mu) \Rightarrow \gamma'(\mu) = \gamma(\mu)) \\ \quad \wedge (\gamma'_1(\mu) = \gamma(\mu) \wedge \gamma'_2(\mu) \neq \gamma(\mu) \Rightarrow \gamma'(\mu) = \gamma'_2(\mu)) \\ \quad \wedge (\gamma'_1(\mu) \neq \gamma(\mu) \wedge \gamma'_2(\mu) = \gamma(\mu) \Rightarrow \gamma'(\mu) = \gamma'_1(\mu)) \\ \quad \wedge (\gamma'_1(\mu) \neq \gamma(\mu) \wedge \gamma'_2(\mu) \neq \gamma(\mu) \Rightarrow \gamma'(\mu) = \gamma'_1(\mu) \vee \gamma'(\mu) = \gamma'_2(\mu))\} \\ \qquad \text{if } \gamma_1 = \gamma_2 = \gamma \end{cases}$$

Figure 2.3 provides an illustration of this definition. Given the two actions α_1 and α_2 depicted by the figure, we obtain the set of merged actions of α_1 and α_2 by the following way. Each action of this set is either α_1, α_2, or a merged action in which each member is modified as by α_1 or by α_2. For example, the two left members of γ_3 to γ_6 are modified to 2 since each of them is modified only by α_1 or only by α_2 while right members are modified in two different ways by these two actions (that leads to 4 different merged actions).

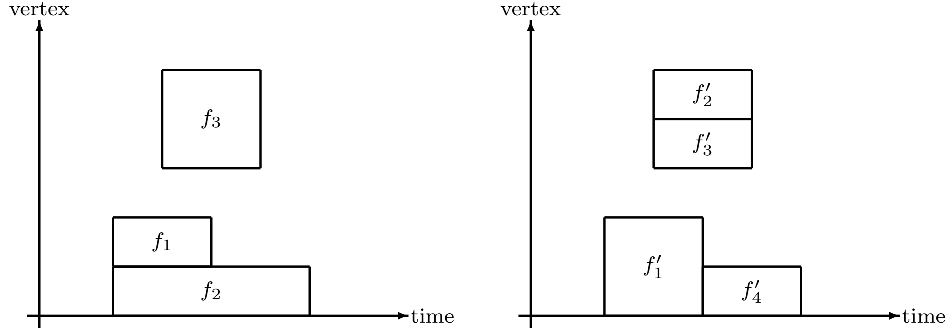

Figure 2.4: Fault patterns $FP = \{f_1, f_2, f_3\}$ and $FP' = \{f'_1, f'_2, f'_3, f'_4\}$ describe the same set of faults if all considered faults have the same behavior.

We can now define the behavior of a system subject to several faults as the merged set of actions of the behavior of each fault. Given two sets of actions S_1 and S_2, the merged set of actions of S_1 and S_2 is the union of merged actions of each action of S_1 with each action of S_2.

Definition 2.9 (*Merged set of actions*)

Given a communication graph g and two sets of actions of g $S_1 \in P(\tilde{A})$ and $S_2 \in P(\tilde{A})$, the merged set of actions of S_1 and S_2 denoted by $S_1 \otimes S_2$ is defined by:

$$S_1 \otimes S_2 = \bigcup_{\substack{\alpha_1 \in S_1 \\ \alpha_2 \in S_2}} \alpha_1 \otimes \alpha_2$$

Before stating the formal definition of an execution subject to a fault pattern, we need to introduce some notations.

For any fault pattern FP, we define the following set of fault patterns:

$$\forall i \in \mathbb{N}, FP(i) = \{f \in FP | b_f \leq i < e_f\}$$

Intuitively, $FP(i)$ denotes the set of faults that are active at time i (*i.e.* that begins before i and that finishes after i).

We define the following notations:

$$\forall i \in \mathbb{N}, V_{FP(i)} = \bigcup_{f \in FP_i} V_f$$
$$\forall i \in \mathbb{N}, \pi_{FP(i)} = \bigotimes_{f \in FP_i} \pi_f$$

Intuitively, $V_{FP(i)}$ denotes the set of faulty vertices at time i and $\pi_{FP(i)}$ denotes the behavior of active faults at time i. We say that a vertex v is correct at configuration γ_i if $v \notin V_{FP(i)}$.

We can now formally define an execution subject to a fault pattern. Each action of such an execution is:

- either an action of the protocol if there is no active fault during this action in the fault pattern,
- or an action of the merged set of actions of faulty behavior for the communication subgraph induced by faulty vertices and an action of the protocol for the communication subgraph induced by non-faulty vertices otherwise.

Definition 2.10 (*Execution subject to a fault pattern*)

Given a communication graph g, an execution $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\dots$ of a distributed protocol π is subject to a fault pattern FP if:

$$\forall i \in \mathbb{N}, \begin{cases} FP(i) = \emptyset \\ \qquad \Rightarrow (\gamma_i, \gamma_{i+1}) \in \pi \\ FP(i) \neq \emptyset \wedge \exists v \in V \setminus V_{FP(i)}, v \in Act(\gamma_i, \gamma_{i+1}) \\ \qquad \Rightarrow (\gamma_i, \gamma_{i+1})\big|_{V \setminus V_{FP(i)}} \in \pi\big|_{V \setminus V_{FP(i)}} \\ FP(i) \neq \emptyset \wedge \exists v \in V_{FP(i)}, v \in Act(\gamma_i, \gamma_{i+1}) \\ \qquad \Rightarrow (\gamma_i, \gamma_{i+1})\big|_{V_{FP(i)}} \in \pi_{FP(i)} \end{cases}$$

Observations on fault patterns The generality of our definition of a fault pattern allows us to describe any abnormal behavior of a distributed system but we can make some observations about our definitions.

First, note that the description of a set of faults by a fault pattern is not unique. Indeed, two fault patterns can describe the same set of faults and be different (see example of Figure 2.4).

On the other hand, that implies that, given an execution subject to an unknown fault pattern, there exists several fault patterns that may produce this execution. For example, we cannot distinguish, as external observers, the following situations: a vertex is faulty but not activated and a vertex is not faulty but no activated.

In our case, such ambiguities introduced by the definition of fault patterns is not a drawback since our goal is to design distributed protocols tolerant to a class of fault patterns (that share some common characteristics) and not to a given and predefined fault pattern.

Modeling of classical fault patterns We illustrate our definition of fault pattern by providing the modeling of some classical fault model.

A fault pattern is transient if it contains only transient faults that end before a given point of the execution. More precisely, a fault pattern FP is a k-transient fault pattern if and only if:

$$\forall f \in FP, e_f \leq k$$

A fault pattern is a permanent crash fault pattern if there are only permanent crash faults that affect at most a given number of vertices and that begin after a given point of the execution. In other words, a fault pattern FP is a (t, ℓ)-permanent

crash fault pattern if and only:

$$\begin{cases} \forall i \in \mathbb{N}, \left|V_{FP(i)}\right| \leq t \\ \forall f \in FP, b_f \geq \ell \wedge e_f = \infty \wedge \pi_f = \{(\gamma,\gamma)\big|_{V_f} | \gamma \in \Gamma\} \end{cases}$$

In a similar way, we can define a permanent Byzantine fault pattern as a fault pattern where there are only Byzantine faults that affect at most a given number of vertices and that begin after a given point of the execution. Formally, a fault pattern FP is a (t,ℓ)-permanent Byzantine fault pattern if and only if:

$$\begin{cases} \forall i \in \mathbb{N}, \left|V_{FP(i)}\right| \leq t \\ \forall f \in FP, b_f \geq \ell \wedge e_f = \infty \wedge \pi_f = (\tilde{A} \setminus \pi)\big|_{V_f} \end{cases}$$

A fault pattern is intermittent if affected vertices are infinitely often strike by faults of the fault pattern after a given point of the execution (note that permanent fault patterns are a particular case of intermittent fault patterns). A fault pattern FP is a (t,ℓ)-intermittent Byzantine fault pattern if and only if:

$$\begin{cases} \exists V' \subseteq V, |V'| \leq t \wedge \forall i \in \mathbb{N}, \begin{cases} V_{FP(i)} \subseteq V' \\ \forall v \in V_{FP(i)}, \exists j > i, v \in V_{FP(j)} \end{cases} \\ \forall f \in FP, b_f \geq \ell \wedge \pi_f = (\tilde{A} \setminus \pi)\big|_{V_f} \end{cases}$$

Composite fault patterns The aim of this thesis is to study distributed systems subject to composite fault patterns, that is fault patterns that gather several classical fault patterns. For example, we study distributed systems simultaneously subject to a transient fault pattern and a permanent crash fault pattern in Part II and subject to a transient fault pattern and an intermittent Byzantine fault pattern in Parts III and IV.

In the following, we provide some definitions of composite fault patterns:

- FP is a (k,t,ℓ)-transient and permanent crash fault pattern if there exists a partition of FP in FP_1 and FP_2 such that FP_1 is a k-transient fault pattern and FP_2 is a (t,ℓ)-permanent crash fault pattern.
- FP is a (k,t,ℓ)-transient and permanent Byzantine fault pattern if there exists a partition of FP in FP_1 and FP_2 such that FP_1 is a k-transient fault pattern and FP_2 is a (t,ℓ)-permanent Byzantine fault pattern.
- FP is a (k,t,ℓ)-transient and intermittent Byzantine fault pattern if there exists a partition of FP in FP_1 and FP_2 such that FP_1 is a k-transient fault pattern and FP_2 is a (t,ℓ)-intermittent Byzantine fault pattern.

Conclusion Our set of definitions allows us to describe any possible fault in our generic computational model. However, the goal of this fault model is to characterize fault tolerance of distributed protocols. To be useful, a fault-tolerant distributed protocol must be resilient to a (as large as possible) class of fault patterns rather

than a single fault. Therefore, we introduced a classification of fault patterns to characterize fault-tolerance properties of distributed protocols. For example, we will see in Section 4.1 that self-stabilizing distributed protocols tolerate any k-transient fault pattern.

The goal of this thesis is to study distributed protocols tolerant to any (k, t, ℓ)-transient and permanent crash fault pattern (Part II) or to any (k, t, ℓ)-transient and intermittent Byzantine fault pattern (Parts III and IV).

CHAPTER 3

Taxonomy of Daemons

> Time is that quality of nature which keeps events from happening all at once. Lately it doesn't seem to be working.
>
> Anonymous

Contents

3.1 Characterization of Daemons **30**
  3.1.1 Distribution 30
  3.1.2 Fairness 31
  3.1.3 Boundedness 35
  3.1.4 Enabledness 37
3.2 Comparing Daemons **41**
  3.2.1 Comparing daemon classes 42
  3.2.2 Preserving execution properties 44
  3.2.3 The Case of the Synchronous Daemon 48
  3.2.4 A map of classical daemons 50
3.3 Daemon Transformers **51**

Daemon (defined in Section 2.3.1) are one of the most central yet less understood concepts in self-stabilization. Indeed, self-stabilizing protocols have to fight against two main adversaries that are interdependent. The first adversary is the initial arbitrary configuration. The second adversary is the amount of asynchrony amongst vertices. In classical fault-tolerant (*e.g.* crash fault tolerant) distributed systems, more asynchrony usually means more impossibilities [FLP85]. In self-stabilization, more synchrony can also be the source of more impossibilities.

Consider for example the mutual exclusion distributed protocol proposed by Herman [Her90] and depicted in Figure 3.1. The distributed protocol operates under the assumption that the communication graph is a unidirectional ring (vertices may only obtain information from their predecessor on the ring, and send information on their successor on the ring). Vertices may hold tokens depending on their initial state, and the goal of the distributed protocol is to ensure that regardless of the initial configuration, the distributed system converges to a point where a single token is present and circulates infinitely often thereafter. Informally, the distributed protocol can be described as follows: whenever a vertex holds a token, it keeps the token with probability p, and sends the token to its immediate successor on the

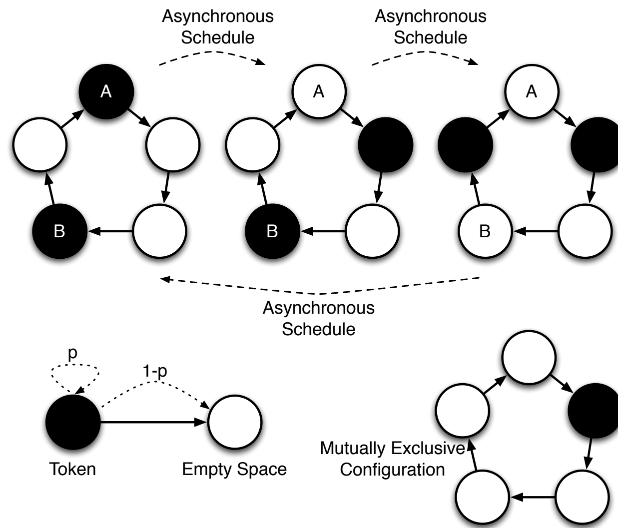

Figure 3.1: Mutual exclusion *vs.* asynchronous scheduling

ring with probability $1 - p$. If a vertex holding a token receives a token from its predecessor, the two tokens are merged. This distributed protocol was well studied assuming synchronous scheduling for all vertices [DHT04, FMP06] and convergence to a single token configuration is expected in $\Theta(n^2)$ time units. Now, if vertex scheduling can be asynchronous, the protocol may not self-stabilize, *i.e.* there may exist an initial configuration and a particular schedule that prevent tokens from merging. Such an example is presented in Figure 3.1: Consider that there exists two initial token in a ring of size five at positions A and B. The scheduling is as follows: the vertex at position A is scheduled for execution until it passes its token (this happens in $O(1)$ expected time), then the vertex at position B is scheduled for execution until it passes its token (again, this happens in $O(1)$ expected time). The new configuration is isomorphic to the first one, and the schedule repeats. As a result, the two initial tokens never merge, and the distributed protocol does not stabilize.

Another example is the vertex coloring distributed protocol of Potop-Butucaru and Tixeuil [PBT00] that is depicted in Figure 3.2. This distributed protocol operates on arbitrary communication graphs under the assumption that no two neighboring vertices are scheduled simultaneously. The distributed protocol colors the communication graph using $deg(g) + 1$ colors in a greedy manner (recall that $deg(g)$ denotes the maximum degree of the communication graph g). Whenever a vertex is scheduled for execution, it checks whether its color conflicts with one of its neighbors (*i.e.* it has the same color as at least one neighbor). If so, it takes the minimal (assuming arbitrary global order on colors) available color to recolor itself. When the scheduling precludes neighbors to be simultaneously activated, the distributed protocol converges to a vertex coloring of the communication graph. When the scheduling is synchronous, the distributed protocol may not stabilize. Consider

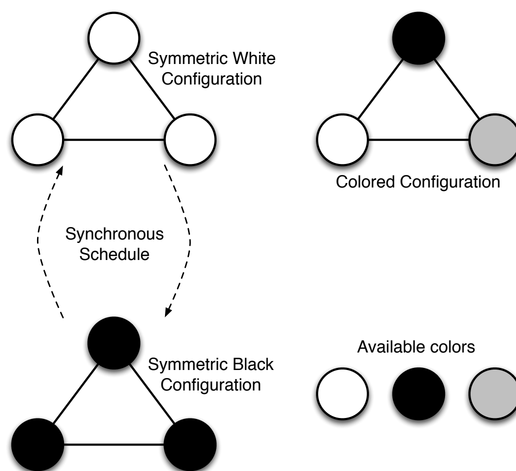

Figure 3.2: Vertex coloring *vs.* synchronous scheduling

the example presented in Figure 3.2: the initial configuration is symmetric white, that is, all vertices have white color. If all vertices are scheduled for execution in this context, they all choose the minimal available color (here, black) and the distributed system reaches a symmetric black configuration. Again, if all vertices are scheduled for execution in this context, they all choose the minimal available color (here, white) and the distributed system reaches a symmetric white configuration. The scheduling repeats and the distributed protocol never stabilizes.

Those two examples are representative of the assumptions made to ensure stabilization of particular distributed protocols. They also show that depending on the problem to be solved, depending on the distributed protocol used to solve the problem, the class of scheduling hypotheses made is quite different. It is nevertheless appealing yet difficult to relate those two scheduling assumptions in a common framework (one relates to temporal constraints, while the other relates to spatial constraints). Literature presenting self-stabilizing distributed protocols typically abstract scheduling assumptions under the notion of daemon. Intuitively, a daemon is just a predicate on global executions, that could in principle be any possible predicate. Recall that we already provide the formal definition of daemon in Section 2.3.1. If every execution of a particular distributed protocol that satisfies the daemon's properties converges to a legitimate configuration, the protocol is self-stabilizing under this daemon.

This approach has the advantage of clearly separating the distributed protocol (that is designed to solve a particular problem) and the scheduling assumptions (that can be seen as an adversary of the distributed protocol, hence the term daemon). However, the problem of comparing possibly unrelated daemons may occur *e.g.* when choosing a particular distributed protocol for implementation in a particular environment (*i.e.* assuming a particular daemon). One would generally like to design a distributed protocol for the strongest adversary (that is, the most inclusive

defining predicate), while impossibility results should be given for the weakest adversary (that is, the least inclusive defining predicate). Obviously, checking whether a particular solution supports a particular environment (that is, the daemon supported by the solution includes the daemon defining the target environment) or whether a particular problem is solvable in a particular environment (that is, the daemon that makes the problem impossible to solve intersects with the daemon defining the target environment) are important questions to which a self-stabilizing protocol designer or implementer should be able to answer.

This chapter presents a taxonomy of daemons already used in the self-stabilizing literature. We review in Section 3.1 the four characteristic traits of daemons existing in the literature. In Section 3.2, we show how our taxonomy can be used to compare daemons in particular contexts with a "more powerful" relation, and maps classical daemons according to their respective power. Section 3.3 reviews algorithms transformations for turning a daemon into another and depicts the influence of the transformation with respect to all four characteristic daemon traits. From now, we consider a distributed system defined by the generic computational model defined in Section 2.3.1.

3.1 Characterization of Daemons

In this section, we review the four characteristic traits of daemons existing in the literature, namely distribution (Section 3.1.1), fairness (Section 3.1.2), boundedness (Section 3.1.3), and enabledness (Section 3.1.4).

3.1.1 Distribution

Constraints about the spatial scheduling of vertices appeared since the seminal paper of Dijkstra [Dij74], as both the central (a single vertex is scheduled for execution at any given action) and the distributed (any subset of enabled vertices may be scheduled for execution at any given action) daemons are presented. Subsequent literature [DPBT00, KY02, DNT09] enriched the initial model with intermediate steps. Intuitively a daemon is k-central if no two vertices less than k hops away are allowed to be simultaneously scheduled. A formal definition follows.

Definition 3.1

Given a communication graph g, a daemon d is k-central if and only if

$$\exists k \in \mathbb{N}, \forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in d(\pi), \forall i \in \mathbb{N}, \forall (u, v) \in V^2,$$
$$[u \neq v \wedge u \in Act(\gamma_i, \gamma_{i+1}) \wedge v \in Act(\gamma_i, \gamma_{i+1})] \Rightarrow dist(g, u, v) > k$$

The set of k-central daemons is denoted by k-$\mathcal{C}$.

In the literature, a 0-central daemon is often called distributed, and a $diam(g)$-central daemon is either called central or sequential.

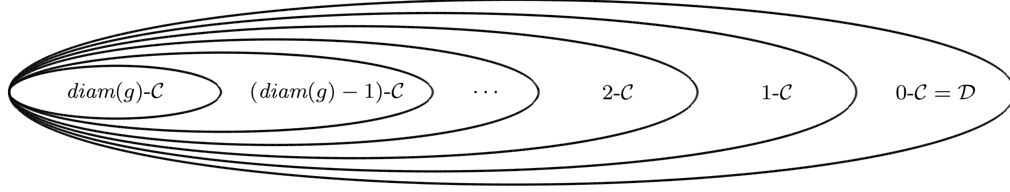

Figure 3.3: Inclusions of sets of daemons with respect to distribution.

Proposition 3.1

Given a communication graph g, we have:

$$\forall k \in \{0, \ldots, diam(g) - 1\}, (k+1)\text{-}\mathcal{C} \subsetneq k\text{-}\mathcal{C}$$

Proof: Let g be a communication graph and $k \in \{0, \ldots, diam(g) - 1\}$. We first prove that $(k+1)$-$\mathcal{C} \subseteq k$-$\mathcal{C}$.

Let d be a daemon such that $d \in (k+1)$-$\mathcal{C}$. Then, by definition, we have:

$$\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in d(\pi), \forall i \in \mathbb{N}, \forall (u, v) \in V^2,$$
$$[u \neq v \wedge u \in Act(\gamma_i, \gamma_{i+1}) \wedge v \in Act(\gamma_i, \gamma_{i+1})] \Rightarrow dist(g, u, v) > k+1$$

As we have $k < k+1$, we obtain that: $\forall (u, v) \in V^2, dist(g, u, v) > k+1 \Rightarrow dist(g, u, v) > k$. As a consequence, we have:

$$\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in d(\pi), \forall i \in \mathbb{N}, \forall (u, v) \in V^2,$$
$$[u \neq v \wedge u \in Act(\gamma_i, \gamma_{i+1}) \wedge v \in Act(\gamma_i, \gamma_{i+1})] \Rightarrow dist(g, u, v) > k$$

By definition, this implies that $d \in k$-$\mathcal{C}$ and shows us that $(k+1)$-$\mathcal{C} \subseteq k$-$\mathcal{C}$.

Then, we prove that $(k+1)$-$\mathcal{C} \neq k$-$\mathcal{C}$. To reach this goal, it is sufficient to construct a daemon d such that: $d \in k$-$\mathcal{C}$ and $d \notin (k+1)$-$\mathcal{C}$.

Let d be a daemon of k-$\mathcal{C}$ that satisfies the following property:

$$\exists \pi \in \Pi, \exists \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in d(\pi), \exists i \in \mathbb{N}, \exists (u, v) \in V^2,$$
$$u \neq v \wedge u \in Act(\gamma_i, \gamma_{i+1}) \wedge v \in Act(\gamma_i, \gamma_{i+1}) \wedge dist(g, u, v) = k+1$$

Note that d exists since the execution σ is not contradictory with the fact that $d \in k$-$\mathcal{C}$. On the other hand, we can observe that $d \notin (k+1)$-$\mathcal{C}$ since the execution σ cannot satisfy the definition of an execution allowed by a $(k+1)$-central daemon. This finishes the proof of the proposition. ■

Figure 3.3 renders Proposition 3.1 graphically.

3.1.2 Fairness

The fairness properties of daemons was not discussed in the seminal paper of Dijkstra [Dij74], as "executing and action" was tantamount to "using critical section" in its mutual exclusion schemes. So, only global progress was assumed, *i.e.* any set

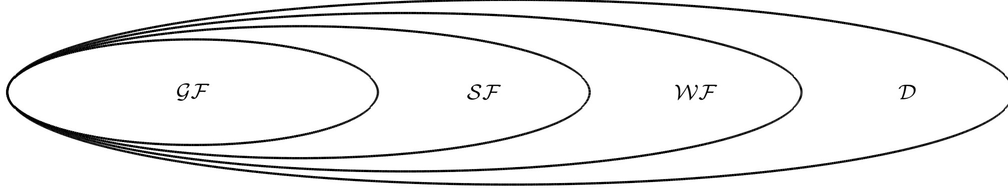

Figure 3.4: Inclusions of sets of daemons with respect to fairness.

of enabled vertices could be scheduled for execution. This very weak assumption was later referred to as an "unfair" daemon [KY97, DPBT00, KY02, DPBT04], since it may happen that a continuously enabled vertex is never scheduled for execution. In our taxonomy, this "unfair" property is simply having no assumptions besides "distributed". The notion of weak fairness [Kar01, HLCW10] prevent such behaviors, as it mandates continuously enabled vertices to eventually be scheduled by the daemon. A formal definition follows.

Definition 3.2

Given a communication graph g, a daemon d is weakly fair if and only if

$$\begin{aligned}&\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots \in \Sigma_\pi,\\ &\qquad [\exists i \in \mathbb{N}, \exists v \in V, (\forall j \geq i, v \in \mathit{Ena}(\gamma_j, \pi)) \wedge (\forall j \geq i, v \notin \mathit{Act}(\gamma_j, \gamma_{j+1}))]\\ &\qquad \Rightarrow \sigma \notin d(\pi)\end{aligned}$$

A weakly fair daemon is also called a fair daemon. The set of (weakly) fair daemons is denoted by $\mathcal{WF}$ or by $\mathcal{F}$. A daemon which is not fair is called unfair. The set of unfair daemons is denoted by $\bar{\mathcal{F}}$ ($\bar{\mathcal{F}} = \mathcal{D} \setminus \mathcal{F}$).

For some distributed protocols (including distributed protocols involving Byzantine behaviors [DPBNT10, DPBT11]), weak fairness is not sufficient to guarantee convergence, and the notion of strong fairness was defined [KC98, Kar01]. Intuitively a daemon is strongly fair if any vertex that is enabled infinitely often is eventually scheduled for execution by the daemon. A formal definition follows.

Definition 3.3

Given a communication graph g, a daemon d is strongly fair if and only if

$$\begin{aligned}&\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots \in \Sigma_\pi,\\ &\quad [\exists i \in \mathbb{N}, \exists v \in V, (\forall j \geq i, \exists k \geq j, v \in \mathit{Ena}(\gamma_k, \pi)) \wedge (\forall j \geq i, v \notin \mathit{Act}(\gamma_j, \gamma_{j+1}))]\\ &\quad \Rightarrow \sigma \notin d(\pi)\end{aligned}$$

The set of strongly fair daemons is denoted by $\mathcal{SF}$.

The strongest notion of fairness (in distributed systems of finite size) is due to Gouda [Gou01]. In short, a weakly stabilizing protocol (*i.e.* a protocol such that from any initial configuration, there exists an execution that leads to a legitimate configuration, see Section 4.1.1) is in fact self-stabilizing assuming Gouda's notion of

fairness. Intuitively, a daemon is *Gouda fair* if from any configuration that appears infinitely often in an execution, every action is eventually scheduled for execution. A formal definition follows.

Definition 3.4

Given a communication graph g, a daemon d is Gouda fair if and only if

$$\begin{array}{l}\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots \in \Sigma_\pi, \forall (\gamma, \gamma') \in \pi \\ \quad [\exists i \in \mathbb{N}, (\forall j \geq i, \exists k \geq j, \gamma_k = \gamma) \wedge (\forall j \geq i, (\gamma_j, \gamma_{j+1}) \neq (\gamma, \gamma'))] \\ \quad \Rightarrow \sigma \notin d(\pi)\end{array}$$

The set of Gouda fair daemons is denoted by $\mathcal{GF}$.

Proposition 3.2

Given a communication graph g, we have:

$$\begin{array}{rcl}\mathcal{GF} & \subsetneq & \mathcal{SF} \\ \mathcal{SF} & \subsetneq & \mathcal{WF} \\ \mathcal{WF} & \subsetneq & \mathcal{D}\end{array}$$

Proof: Firstly, we prove that $\mathcal{GF} \subsetneq \mathcal{SF}$. We begin by proving that $\mathcal{GF} \subseteq \mathcal{SF}$.

Let d be a daemon of $\mathcal{GF}$. Assume that we have $\pi \in \Pi$ and $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots \in \Sigma_\pi$ such that

$$\exists i \in \mathbb{N}, \exists v \in V, (\forall j \geq i, \exists k \geq j, v \in Ena(\gamma_k, \pi)) \wedge (\forall j \geq i, v \notin Act(\gamma_j, \gamma_{j+1}))$$

Since π is a finite subset of actions of g, this property implies the following:

$$\exists (\gamma, \gamma') \in \pi, \exists i \in \mathbb{N}, (\forall j \geq i, \exists k \geq j, \gamma_k = \gamma) \wedge (\forall j \geq i, (\gamma_j, \gamma_{j+1}) \neq (\gamma, \gamma'))$$

As $d \in \mathcal{GF}$, we can deduce that $\sigma \notin d(\pi)$ by definition. Consequently, we have:

$$\begin{array}{l}\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots \in \Sigma_\pi, \\ \quad [\exists i \in \mathbb{N}, \exists v \in V, (\forall j \geq i, \exists k \geq j, v \in Ena(\gamma_k, \pi)) \wedge (\forall j \geq i, v \notin Act(\gamma_j, \gamma_{j+1}))] \\ \quad \Rightarrow \sigma \notin d(\pi)\end{array}$$

This proves that $d \in \mathcal{SF}$ and hence that $\mathcal{GF} \subseteq \mathcal{SF}$.

It remains to prove that $\mathcal{GF} \neq \mathcal{SF}$. It is sufficient to construct a daemon d such that $d \in \mathcal{SF}$ and $d \notin \mathcal{GF}$.

Let g be a communication graph and π be a distributed protocol such that:

$$\exists (\gamma, \gamma', \gamma'') \in \Gamma^3, (\gamma, \gamma') \in \pi \wedge (\gamma, \gamma'') \in \pi \wedge Act(\gamma, \gamma') = Act(\gamma, \gamma'')$$

Then, it is possible to define a daemon $d \in \mathcal{SF}$ and an execution $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots \in \Sigma_\pi$ such that:

$$\exists i \in \mathbb{N}, (\forall j \geq i, \exists k \geq j, \gamma_k = \gamma) \wedge (\forall j \geq i, \gamma_j = \gamma \Rightarrow (\gamma_j, \gamma_{j+1}) = (\gamma, \gamma'') \neq (\gamma, \gamma'))$$

We can conclude that $d \notin \mathcal{GF}$ since the execution σ cannot satisfy the definition of an execution allowed by a Gouda fair daemon. That proves the result (since $d \in \mathcal{SF}$ by assumption).

Now, we prove that $\mathcal{SF} \subsetneq \mathcal{WF}$. We first prove that $\mathcal{SF} \subseteq \mathcal{WF}$.

Let d be a daemon of $\mathcal{SF}$. Assume that we have $\pi \in \Pi$ and $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in \Sigma_\pi$ such that

$$\exists i \in \mathbb{N}, \exists v \in V, (\forall j \geq i, v \in Ena(\gamma_j, \pi)) \wedge (\forall j \geq i, v \notin Act(\gamma_j, \gamma_{j+1}))$$

This property implies the following:

$$\exists i \in \mathbb{N}, \exists v \in V, (\forall j \geq i, \exists k = j, v \in Ena(\gamma_k, \pi)) \wedge (\forall j \geq i, v \notin Act(\gamma_j, \gamma_{j+1}))$$

As $d \in \mathcal{SF}$, we can deduce that $\sigma \notin d(\pi)$ by definition. Consequently, we have:

$$\begin{aligned}&\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in \Sigma_\pi,\\ &\qquad [\exists i \in \mathbb{N}, \exists v \in V, (\forall j \geq i, v \in Ena(\gamma_j, \pi)) \wedge (\forall j \geq i, v \notin Act(\gamma_j, \gamma_{j+1}))]\\ &\qquad \Rightarrow \sigma \notin d(\pi)\end{aligned}$$

This proves that $d \in \mathcal{WF}$ and hence that $\mathcal{SF} \subseteq \mathcal{WF}$.

It remains to prove that $\mathcal{SF} \neq \mathcal{WF}$. It is sufficient to construct a daemon d such that $d \in \mathcal{WF}$ and $d \notin \mathcal{SF}$.

Let g be a communication graph, π be a distributed protocol and u, v be two vertices such that:

$$\exists (\gamma, \gamma') \in \Gamma^2, \begin{cases} v \in Ena(\gamma, \pi) \wedge u \in Ena(\gamma, \pi) \wedge v \notin Ena(\gamma', \pi) \wedge u \in Ena(\gamma', \pi)\\ v \notin Act(\gamma, \gamma') \wedge u \in Act(\gamma, \gamma') \wedge v \notin Act(\gamma', \gamma) \wedge u \in Act(\gamma', \gamma)\\ (\gamma, \gamma') \in \pi \wedge (\gamma', \gamma) \in \pi \end{cases}$$

Then, it is possible to define a daemon $d \in \mathcal{WF}$ and an execution $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in \Sigma_\pi$ such that:

$$\sigma \in d(\pi) \wedge (\forall p \in \mathbb{N}, \gamma_{2p} = \gamma \wedge \gamma_{2p+1} = \gamma')$$

We can observe that σ satisfies the following property:

$$\begin{aligned}&\exists i = 0 \in \mathbb{N}, [(\forall j \geq i, (\exists k \geq j, v \in Ena(\gamma_k, \pi)) \wedge (\exists k' \geq j, v \notin Ena(\gamma_{k'}, \pi))) \wedge\\ &\qquad (\forall j \geq i, v \notin Act(\gamma_j, \gamma_{j+1}))]\end{aligned}$$

We can conclude that $d \notin \mathcal{SF}$ since the execution σ cannot satisfy the definition of an execution allowed by a strongly fair daemon. That proves the result (since $d \in \mathcal{WF}$ by assumption).

Finally, we prove that $\mathcal{WF} \subsetneq \mathcal{D}$. As the definition implies that $\mathcal{WF} \subseteq \mathcal{D}$, it remains to prove that $\mathcal{WF} \neq \mathcal{D}$. It is sufficient to construct a daemon d such that $d \in \mathcal{D}$ and $d \notin \mathcal{WF}$.

Let g be a communication graph and π be a distributed protocol such that there exists $v \in V$ satisfying:

$$\forall (\gamma, \gamma') \in \pi, v \in Ena(\gamma, \pi) \Rightarrow |Ena(\gamma, \pi)| \geq 2$$

Then, it is possible to define a daemon d and an execution $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in$

Σ_π such that:

$$\forall i \in \mathbb{N}, v \notin Act(\gamma_i, \gamma_{i+1}) \wedge \sigma \in d(\pi)$$

We can conclude that $d \notin \mathcal{WF}$ since the execution σ cannot satisfy the definition of an execution allowed by a weakly fair daemon. That proves the result (since $d \in \mathcal{D}$ by definition). ■

Figure 3.4 renders Proposition 3.2 graphically. Devismes *et al.* [DTY08] observe that in infinite distributed systems, Gouda fairness is not the strongest form of fairness.

3.1.3 Boundedness

Boundedness was first presented in [BCD95] and later refined in [DPBT00] as a property achieved by a daemon transformer (see also Section 3.3) and was also used as a benchmark to evaluate the performance of self-stabilizing distributed protocols under various kinds of daemons [BDPBM02, BPBJ01]. Intuitively a daemon is k-bounded if no vertex can be scheduled more than k times between any two consecutive scheduling of any other vertex. Note that this does not imply that there exists a bound on the "speed" ratio between any two vertices: in particular if a vertex is never scheduled in a particular execution, another vertex may be scheduled more than k times in the execution sequel without breaking the k-boundedness constraint. As a matter of fact, a daemon can be both k-bounded and unfair. A formal definition follows.

Definition 3.5

Given a communication graph g, a daemon d is k-bounded if and only if

$$\begin{aligned}
&\exists k \in \mathbb{N}^*, \forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots \in d(\pi), \forall (i,j) \in \mathbb{N}^2, \forall v \in V,\\
&\quad \big[[v \in Act(\gamma_i, \gamma_{i+1}) \wedge (\forall \ell \in \mathbb{N}, \ell < i \Rightarrow v \notin Act(\gamma_\ell, \gamma_{\ell+1}))]\\
&\quad \Rightarrow \forall u \in V \setminus \{v\}, |\{\ell \in \mathbb{N} | \ell < i \wedge u \in Act(\gamma_\ell, \gamma_{\ell+1})\}| \leq k\big] \wedge\\
&\quad \big[[i < j \wedge v \in Act(\gamma_i, \gamma_{i+1}) \wedge v \in Act(\gamma_j, \gamma_{j+1}) \wedge\\
&\quad (\forall \ell \in \mathbb{N}, i < \ell < j \Rightarrow v \notin Act(\gamma_\ell, \gamma_{\ell+1}))]\\
&\quad \Rightarrow \forall u \in V \setminus \{v\}, |\{\ell \in \mathbb{N} | i \leq \ell < j \wedge u \in Act(\gamma_\ell, \gamma_{\ell+1})\}| \leq k\big]
\end{aligned}$$

The set of k-bounded daemons is denoted by k-$\mathcal{B}$. The set of bounded daemons is denoted by $\mathcal{B}$ ($\mathcal{B} = \bigcup_{k \in \mathbb{N}^*} k\text{-}\mathcal{B}$). A daemon that is not k-bounded for any $k \in \mathbb{N}^*$ is called unbounded. The set of unbounded daemons is denoted by $\bar{\mathcal{B}}$ ($\bar{\mathcal{B}} = \mathcal{D} \setminus \mathcal{B}$).

Proposition 3.3

Given a communication graph g, we have:

$$\forall k \in \mathbb{N}^*, \begin{cases} k\text{-}\mathcal{B} \subsetneq (k+1)\text{-}\mathcal{B} \\ k\text{-}\mathcal{B} \subsetneq \mathcal{D} \end{cases}$$

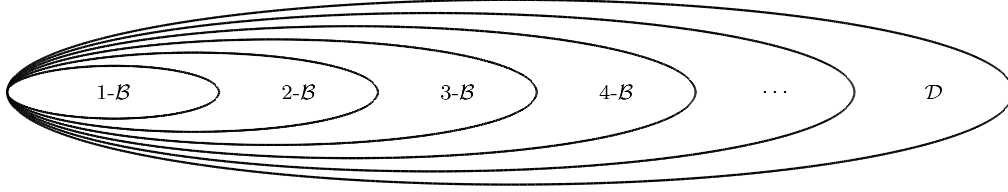

Figure 3.5: Inclusions of sets of daemons with respect to boundedness.

Proof: Let g be a communication graph and $k \in \mathbb{N}^*$. We first prove that k-$\mathcal{B} \subseteq (k+1)$-$\mathcal{B}$.

Let d be a daemon such that $d \in k$-$\mathcal{B}$. Then, by definition, we have:

$$\begin{array}{l}\exists k \in \mathbb{N}^*, \forall \pi \in \Pi, \forall \sigma = (\gamma_0,\gamma_1)(\gamma_1,\gamma_2)\ldots \in d(\pi), \forall (i,j) \in \mathbb{N}^2, \forall v \in V,\\ \quad \big[[v \in Act(\gamma_i,\gamma_{i+1}) \wedge (\forall \ell \in \mathbb{N}, \ell < i \Rightarrow v \notin Act(\gamma_\ell,\gamma_{\ell+1}))]\\ \quad \Rightarrow \forall u \in V \setminus \{v\}, |\{\ell \in \mathbb{N} | \ell < i \wedge u \in Act(\gamma_\ell,\gamma_{\ell+1})\}| \leq k\big] \wedge\\ \quad \big[[i < j \wedge v \in Act(\gamma_i,\gamma_{i+1}) \wedge v \in Act(\gamma_j,\gamma_{j+1}) \wedge\\ \quad (\forall \ell \in \mathbb{N}, i < \ell < j \Rightarrow v \notin Act(\gamma_\ell,\gamma_{\ell+1}))]\\ \quad \Rightarrow \forall u \in V \setminus \{v\}, |\{\ell \in \mathbb{N} | i \leq \ell < j \wedge u \in Act(\gamma_\ell,\gamma_{\ell+1})\}| \leq k\big]\end{array}$$

As we have $k < k+1$, we obtain that:

$$\begin{array}{l}\exists k \in \mathbb{N}^*, \forall \pi \in \Pi, \forall \sigma = (\gamma_0,\gamma_1)(\gamma_1,\gamma_2)\ldots \in d(\pi), \forall (i,j) \in \mathbb{N}^2, \forall v \in V,\\ \quad \big[[v \in Act(\gamma_i,\gamma_{i+1}) \wedge (\forall \ell \in \mathbb{N}, \ell < i \Rightarrow v \notin Act(\gamma_\ell,\gamma_{\ell+1}))]\\ \quad \Rightarrow \forall u \in V \setminus \{v\}, |\{\ell \in \mathbb{N} | \ell < i \wedge u \in Act(\gamma_\ell,\gamma_{\ell+1})\}| \leq k+1\big] \wedge\\ \quad \big[[i < j \wedge v \in Act(\gamma_i,\gamma_{i+1}) \wedge v \in Act(\gamma_j,\gamma_{j+1}) \wedge\\ \quad (\forall \ell \in \mathbb{N}, i < \ell < j \Rightarrow v \notin Act(\gamma_\ell,\gamma_{\ell+1}))]\\ \quad \Rightarrow \forall u \in V \setminus \{v\}, |\{\ell \in \mathbb{N} | i \leq \ell < j \wedge u \in Act(\gamma_\ell,\gamma_{\ell+1})\}| \leq k+1\big]\end{array}$$

By definition, this implies that $d \in (k+1)$-$\mathcal{B}$ and shows us that k-$\mathcal{B} \subseteq (k+1)$-$\mathcal{B}$.

Now, we must prove that k-$\mathcal{B} \neq (k+1)$-$\mathcal{B}$. To reach this goal, it is sufficient to construct a daemon d such that: $d \in (k+1)$-$\mathcal{B}$ and $d \notin k$-$\mathcal{B}$.

Let d be a daemon of $(k+1)$-$\mathcal{B}$ which satisfies the following property:

$$\begin{array}{l}\exists \pi \in \Pi, \exists \sigma = (\gamma_0,\gamma_1)(\gamma_1,\gamma_2)\ldots \in d(\pi), \exists (i,j) \in \mathbb{N}^2, \exists v \in V,\\ \quad i < j \wedge v \in Act(\gamma_i,\gamma_{i+1}) \wedge v \in Act(\gamma_j,\gamma_{j+1})\\ \quad \wedge (\forall \ell \in \mathbb{N}, i < \ell < j \Rightarrow v \notin Act(\gamma_\ell,\gamma_{\ell+1}))\\ \quad \wedge (\exists u \in V \setminus \{v\}, |\{\ell \in \mathbb{N} | i \leq \ell < j \wedge u \in Act(\gamma_\ell,\gamma_{\ell+1})\}| = k+1)\end{array}$$

Note that d exists since the execution σ is not contradictory with the fact that $d \in (k+1)$-$\mathcal{B}$. On the other hand, we can observe that $d \notin k$-$\mathcal{B}$ since the execution σ cannot satisfy the definition of an execution allowed by a k-bounded daemon. This finishes the proof of the first property.

Finally, we prove that k-$\mathcal{B} \subsetneq \mathcal{D}$. By definition, we have: k-$\mathcal{B} \subseteq \mathcal{D}$, it remains to prove that k-$\mathcal{B} \neq \mathcal{D}$. By the first property, we know that there exists a daemon d such that $d \in (k+1)$-$\mathcal{B}$ and $d \notin k$-$\mathcal{B}$. As $(k+1)$-$\mathcal{B} \subseteq \mathcal{D}$ by definition, we have the result. ■

Figure 3.5 renders Proposition 3.3 graphically.

3.1.4 Enabledness

Enabledness is a characterization of daemon properties that is introduced in this thesis. It is defined to be related to the intuitive notion that the ratio between the "speed" of the fastest vertex and that of the slowest vertex is bounded. In an asynchronous setting where we use configurations and time-independent actions between configurations, k-enabledness intuitively means that a particular vertex cannot be enabled more than k times before being activated. A formal definition follows.

Definition 3.6

Given a communication graph g, a daemon d is k-enabled if and only if

$$\begin{aligned}\exists k \in \mathbb{N}, \forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots \in d(\pi), \forall (i,j) \in \mathbb{N}^2, \forall v \in V,\\ \big[[v \in Act(\gamma_i, \gamma_{i+1}) \wedge (\forall \ell \in \mathbb{N}, \ell < i \Rightarrow v \notin Act(\gamma_\ell, \gamma_{\ell+1}))]\\ \Rightarrow |\{\ell \in \mathbb{N} | \ell < i \wedge v \in Ena(\gamma_\ell, \pi)\}| \leq k\big] \wedge\\ \big[[i < j \wedge v \in Act(\gamma_i, \gamma_{i+1}) \wedge v \in Act(\gamma_j, \gamma_{j+1})\\ \wedge (\forall \ell \in \mathbb{N}, i < \ell < j \Rightarrow v \notin Act(\gamma_\ell, \gamma_{\ell+1}))]\\ \Rightarrow |\{\ell \in \mathbb{N} | i < \ell < j \wedge v \in Ena(\gamma_\ell, \pi)\}| \leq k\big] \wedge\\ \big[[v \in Act(\gamma_i, \gamma_{i+1}) \wedge (\forall \ell \in \mathbb{N}, \ell > i \Rightarrow v \notin Act(\gamma_\ell, \gamma_{\ell+1}))]\\ \Rightarrow |\{\ell \in \mathbb{N} | \ell > i \wedge v \in Ena(\gamma_\ell, \pi)\}| \leq k\big]\end{aligned}$$

The set of k-enabled daemons is denoted by k-$\mathcal{E}$. The set of daemons of bounded enabledness is denoted by $\mathcal{E}$ ($\mathcal{E} = \bigcup_{k \in \mathbb{N}} k\text{-}\mathcal{E}$). A daemon that is not k-enabled for any $k \in \mathbb{N}$ has an unbounded enabledness. The set of daemons of unbounded enabledness is denoted by $\bar{\mathcal{E}}$ ($\bar{\mathcal{E}} = \mathcal{D} \setminus \mathcal{E}$).

Proposition 3.4

Given a communication graph g, we have:

$$\forall k \in \mathbb{N}, \begin{cases} k\text{-}\mathcal{E} \subsetneq (k+1)\text{-}\mathcal{E} \\ k\text{-}\mathcal{E} \subsetneq \mathcal{D} \end{cases}$$

Proof: Let g be a communication graph and $k \in \mathbb{N}$. We first prove that k-$\mathcal{E} \subseteq (k+1)$-$\mathcal{E}$.

Let d be a daemon such that $d \in k$-$\mathcal{E}$. Then, by definition, we have:

$$\begin{aligned}\exists k \in \mathbb{N}, \forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots \in d(\pi), \forall (i,j) \in \mathbb{N}^2, \forall v \in V,\\ \big[[v \in Act(\gamma_i, \gamma_{i+1}) \wedge (\forall \ell \in \mathbb{N}, \ell < i \Rightarrow v \notin Act(\gamma_\ell, \gamma_{\ell+1}))]\\ \Rightarrow |\{\ell \in \mathbb{N} | \ell < i \wedge v \in Ena(\gamma_\ell, \pi)\}| \leq k\big] \wedge\\ \big[[i < j \wedge v \in Act(\gamma_i, \gamma_{i+1}) \wedge v \in Act(\gamma_j, \gamma_{j+1})\\ \wedge (\forall \ell \in \mathbb{N}, i < \ell < j \Rightarrow v \notin Act(\gamma_\ell, \gamma_{\ell+1}))]\\ \Rightarrow |\{\ell \in \mathbb{N} | i < \ell < j \wedge v \in Ena(\gamma_\ell, \pi)\}| \leq k\big] \wedge\\ \big[[v \in Act(\gamma_i, \gamma_{i+1}) \wedge (\forall \ell \in \mathbb{N}, \ell > i \Rightarrow v \notin Act(\gamma_\ell, \gamma_{\ell+1}))]\\ \Rightarrow |\{\ell \in \mathbb{N} | \ell > i \wedge v \in Ena(\gamma_\ell, \pi)\}| \leq k\big]\end{aligned}$$

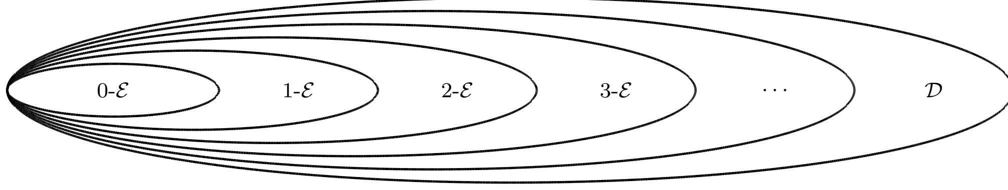

Figure 3.6: Inclusions of sets of daemons with respect to enabledness.

As we have $k < k+1$, we obtain that:

$$\begin{aligned}
\exists k \in \mathbb{N}, \forall \pi \in \Pi, &\forall \sigma = (\gamma_0,\gamma_1)(\gamma_1,\gamma_2)\ldots \in d(\pi), \forall (i,j) \in \mathbb{N}^2, \forall v \in V,\\
&\big[[v \in Act(\gamma_i,\gamma_{i+1}) \wedge (\forall \ell \in \mathbb{N}, \ell < i \Rightarrow v \notin Act(\gamma_\ell,\gamma_{\ell+1}))]\\
&\Rightarrow |\{\ell \in \mathbb{N} | \ell < i \wedge v \in Ena(\gamma_\ell,\pi)\}| \leq k+1\big] \wedge\\
&\big[[i < j \wedge v \in Act(\gamma_i,\gamma_{i+1}) \wedge v \in Act(\gamma_j,\gamma_{j+1})\\
&\wedge (\forall \ell \in \mathbb{N}, i < \ell < j \Rightarrow v \notin Act(\gamma_\ell,\gamma_{\ell+1}))]\\
&\Rightarrow |\{\ell \in \mathbb{N} | i < \ell < j \wedge v \in Ena(\gamma_\ell,\pi)\}| \leq k+1\big] \wedge\\
&\big[[v \in Act(\gamma_i,\gamma_{i+1}) \wedge (\forall \ell \in \mathbb{N}, \ell > i \Rightarrow v \notin Act(\gamma_\ell,\gamma_{\ell+1}))]\\
&\Rightarrow |\{\ell \in \mathbb{N} | \ell > i \wedge v \in Ena(\gamma_\ell,\pi)\}| \leq k+1\big]
\end{aligned}$$

By definition, this implies that $d \in (k+1)$-$\mathcal{E}$ and shows us that k-$\mathcal{E} \subseteq (k+1)$-$\mathcal{E}$.

Now, we must prove that k-$\mathcal{E} \neq (k+1)$-$\mathcal{E}$. To reach this goal, it is sufficient to construct a daemon d such that: $d \in (k+1)$-$\mathcal{E}$ and $d \notin k$-$\mathcal{E}$.

Let d be a daemon of $(k+1)$-$\mathcal{E}$ which satisfies the following property:

$$\begin{aligned}
\exists \pi \in \Pi, \exists \sigma = &(\gamma_0,\gamma_1)(\gamma_1,\gamma_2)\ldots \in d(\pi), \exists (i,j) \in \mathbb{N}^2, \exists v \in V,\\
&i < j \wedge v \in Act(\gamma_i,\gamma_{i+1}) \wedge v \in Act(\gamma_j,\gamma_{j+1})\\
&\wedge (\forall \ell \in \mathbb{N}, i < \ell < j \Rightarrow v \notin Act(\gamma_\ell,\gamma_{\ell+1}))\\
&\wedge |\{\ell \in \mathbb{N} | i < \ell < j \wedge v \in Ena(\gamma_\ell,\pi)\}| = k+1
\end{aligned}$$

Note that d exists since the execution σ is not contradictory with the fact that $d \in (k+1)$-$\mathcal{E}$. On the other hand, we can observe that $d \notin k$-$\mathcal{E}$ since the execution σ cannot satisfy the definition of an execution allowed by a k-enabled daemon. This finishes the proof of the first property.

Finally, we prove that k-$\mathcal{E} \subsetneq \mathcal{D}$. By definition, we have: k-$\mathcal{E} \subseteq \mathcal{D}$, it remains to prove that k-$\mathcal{E} \neq \mathcal{D}$. By the first property, we know that there exists a daemon d such that $d \in (k+1)$-$\mathcal{E}$ and $d \notin k$-$\mathcal{E}$. As $(k+1)$-$\mathcal{E} \subseteq \mathcal{D}$ by definition, we have the result. ■

Figure 3.6 renders Proposition 3.4 graphically. Unlike previous characteristic properties of daemons, enabledness is not completely independent from others. Relationship between enabledness and fairness and boundedness are depicted in the sequel.

Relationship between fairness and enabledness Daemons with bounded enabledness cannot ignore scheduling vertices more than k times, implying that the overall schedule is at least weakly fair. Nevertheless, the following proposition shows that the converse is not true (*i.e.* there exist daemons that are weakly fair but do

not have bounded enabledness, furthermore these daemons are not strongly fair either). There also exist daemons that are strongly fair or Gouda fair, yet do not have finite enabledness.

Proposition 3.5

For any given communication graph g, the following properties are satisfied:

$$\begin{array}{c} \forall d \in \mathcal{D}, d \in \mathcal{E} \Rightarrow d \in \mathcal{WF} \\ \exists d \in \mathcal{WF} \setminus (\mathcal{E} \cup \mathcal{SF}) \\ \exists d \in \mathcal{SF} \setminus (\mathcal{E} \cup \mathcal{GF}) \\ \exists d \in \mathcal{GF} \setminus \mathcal{E} \end{array}$$

Proof: Let g be a communication graph. Let d be a daemon such that $d \in \mathcal{E}$. Then, there exists $k \in \mathbb{N}$ such that $d \in k\text{-}\mathcal{E}$. We are going to prove that $d \in \mathcal{WF}$.

Assume that π is a distributed protocol and $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \dots$ is an execution of $d(\pi)$ satisfying the following property:

$$\begin{array}{r} \exists i \in \mathbb{N}^*, \exists v \in V, v \in Act(\gamma_{i-1}, \gamma_i) \wedge (\forall j \geq i, v \in Ena(\gamma_j, \pi)) \\ \wedge (\forall j \geq i, v \notin Act(\gamma_j, \gamma_{j+1})) \end{array}$$

Then, we have:

$$\begin{array}{l} [v \in Act(\gamma_{i-1}, \gamma_i) \wedge (\forall \ell \in \mathbb{N}, \ell > i \Rightarrow v \notin Act(\gamma_\ell, \gamma_{\ell+1}))] \\ \qquad \wedge |\{\ell \in \mathbb{N} | \ell > i \wedge v \in Ena(\gamma_\ell, \pi)\}| = \infty > k \end{array}$$

This property is contradictory with $\sigma \in d(\pi)$ and $d \in k\text{-}\mathcal{E}$. hence, we deduct that:

$$\begin{array}{l} \forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \dots \in \Sigma_\pi, \\ \qquad [\exists i \in \mathbb{N}, \exists v \in V, (\forall j \geq i, v \in Ena(\gamma_j, \pi)) \wedge (\forall j \geq i, v \notin Act(\gamma_j, \gamma_{j+1}))] \\ \qquad \Rightarrow \sigma \notin d(\pi) \end{array}$$

That means that $d \in \mathcal{WF}$. It follows that: $\forall d \in \mathcal{D}, d \in \mathcal{E} \Rightarrow d \in \mathcal{WF}$.

Now, we prove that $\exists d \in \mathcal{GF} \setminus \mathcal{E}$. To this goal, consider a daemon d such that $d \in \mathcal{GF}$ and a distributed protocol π_1 such that:

$$\exists (\gamma_0, \gamma_1, \gamma_2) \in \Gamma^3, \exists v \in V, \left\{ \begin{array}{l} (\gamma_0, \gamma_1) \in \pi_1 \wedge v \in Ena(\gamma_0, \pi_1) \wedge v \in Act(\gamma_0, \gamma_1) \\ (\gamma_1, \gamma_2) \in \pi_1 \wedge v \in Ena(\gamma_1, \pi_1) \wedge v \notin Act(\gamma_1, \gamma_2) \\ (\gamma_2, \gamma_1) \in \pi_1 \wedge v \in Ena(\gamma_2, \pi_1) \wedge v \notin Act(\gamma_2, \gamma_1) \end{array} \right.$$

Let σ be an execution of $d(\pi_1)$ starting from γ_2. Now, we define the following set of executions of π_1 (where the product operator denotes the concatenation of portions of executions):

$$\forall k \in \mathbb{N}, \sigma_k = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2).\big[(\gamma_2, \gamma_1)(\gamma_1, \gamma_2)\big]^k.\sigma$$

We can define a daemon d' in the following way:

$$\left\{ \begin{array}{l} \forall \pi \in \Pi \setminus \{\pi_1\}, d'(\pi) = d(\pi) \\ d'(\pi_1) = d(\pi_1) \cup \{\sigma_k | k \in \mathbb{N}\} \end{array} \right.$$

Then, we can observe that $d' \in \mathcal{GF}$ by construction and that, for any $k \in \mathbb{N}$, the execution $\sigma_k \in d'(\pi_1)$ does not satisfy the definition of k-enabledness. Consequently, we prove that: $d' \in \mathcal{GF} \setminus \mathcal{E}$. If we follow the same reasoning starting from a daemon d in $\mathcal{WF} \setminus \mathcal{SF}$ (respectively in $\mathcal{SF} \setminus \mathcal{GF}$), we prove that $d' \in \mathcal{WF} \setminus (\mathcal{E} \cup \mathcal{SF})$ (respectively that $d' \in \mathcal{SF} \setminus (\mathcal{E} \cup \mathcal{GF})$), that ends the proof. ∎

Relationship between boundedness and enabledness As previously mentioned, there is not relationship between boundedness and fairness. In this section, we prove that there is a connexion between (finite) enabledness and (finite) boundedness. In particular, if a daemon is both k-enabled and k'-bounded (for some particular integers k and k'), we have $k \leq (n-1) \times k'$ (where n denotes the number of vertices in the system). However, there exist daemons that are k-enabled (for some integer k) but do not have finite boundedness, and daemons that are k'-bounded (for some integer k') but do not have finite enabledness.

Proposition 3.6

For any given communication graph g, we have:

$$\forall d \in \mathcal{D}, \forall (k,k') \in \mathbb{N} \times \mathbb{N}^*, (d \in k\text{-}\mathcal{E} \wedge d \in k'\text{-}\mathcal{B}) \Rightarrow k \leq (n-1) \times k'$$
$$\forall k \in \mathbb{N}, \exists d \in k\text{-}\mathcal{E} \setminus \mathcal{B}$$
$$\forall k \in \mathbb{N}^*, \exists d \in k\text{-}\mathcal{B} \setminus \mathcal{E}$$

Proof: Firstly, we prove that $\forall d \in \mathcal{D}, \forall (k,k') \in \mathbb{N} \times \mathbb{N}^*, (d \in k\text{-}\mathcal{E} \wedge d \in k'\text{-}\mathcal{B}) \Rightarrow k \leq (n-1) \times k'$. In this goal, consider a daemon d such that $d \in k\text{-}\mathcal{E}$ and $d \in k'\text{-}\mathcal{B}$ for two given $(k,k') \in \mathbb{N} \times \mathbb{N}^*$.

As d is k'-bounded, we know by definition that between two consecutive actions of any vertex v, any vertex u such that $u \neq v$ takes at most k' actions. This implies that there exists at most $(n-1) \times k'$ actions between two consecutive actions of v (since the daemon must ensure the progress). This implies that, between two consecutive actions of v, there exists at most $(n-1) \times k'$ configurations where v is enabled (without being activated by construction). As we know that d has a bounded enabledness k, we can deduce that $k \leq (n-1) \times k'$, that proves the result.

Secondly, we prove that $\forall k \in \mathbb{N}, \exists d \in k\text{-}\mathcal{E} \setminus \mathcal{B}$. To this goal, consider $k \in \mathbb{N}$, a daemon d such that $d \in k\text{-}\mathcal{E}$ and a distributed protocol π_1 such that:

$$\forall \ell \in \mathbb{N}^*, \exists (\gamma_{\ell+1}, \gamma_\ell, \dots, \gamma_1, \gamma_0) \in \Gamma^{\ell+2},$$
$$\exists v \in V, \begin{cases} \forall i \in \{0,\dots,\ell\}, (\gamma_{i+1}, \gamma_i) \in \pi_1 \\ \forall i \in \{0,\dots,\ell\}, Act(\gamma_{i+1}, \gamma_i) = Ena(\gamma_{i+1}, \pi_1) = V \end{cases}$$

Let σ be an execution of $d(\pi_1)$ starting from γ_0. Now, we define the following set of executions of π_1 (where the product operator denotes the concatenation of portions of executions):

$$\forall k' \in \mathbb{N}^*, \sigma_{k'} = (\gamma_{k'+1}, \gamma_{k'})(\gamma_{k'}, \gamma_{k'-1}) \dots (\gamma_2, \gamma_1)(\gamma_1, \gamma_0).\sigma$$

Note that, for any $k' \in \mathbb{N}^*$, the portion of execution $(\gamma_{k'+1}, \gamma_{k'})(\gamma_{k'}, \gamma_{k'-1}) \dots (\gamma_2, \gamma_1)(\gamma_1, \gamma_0)$ is 0-enabled. Hence, any execution of $\{\sigma_{k'} | k' \in \mathbb{N}^*\}$ is k-enabled.

We can define a daemon d' in the following way:

$$\begin{cases} \forall \pi \in \Pi \setminus \{\pi_1\}, d'(\pi) = d(\pi) \\ d'(\pi_1) = d(\pi_1) \cup \{\sigma_{k'} | k' \in \mathbb{N}^*\} \end{cases}$$

Then, we can observe that $d' \in k\text{-}\mathcal{E}$ by construction and that, for any $k' \in \mathbb{N}^*$, the execution $\sigma_{k'} \in d'(\pi_1)$ does not satisfy the definition of k'-boundedness. Consequently, we prove that: $d' \in k\text{-}\mathcal{E} \setminus \bigcup_{k' \in \mathbb{N}^*} k'\text{-}\mathcal{B} = k\text{-}\mathcal{E} \setminus \mathcal{B}$.

Finally, we prove that $\forall k \in \mathbb{N}^*, \exists d \in k\text{-}\mathcal{B} \setminus \mathcal{E}$. To this goal, consider $k \in \mathbb{N}^*$, a daemon d such that $d \in k\text{-}\mathcal{B}$ and a distributed protocol π_1 such that:

$$\exists (\gamma_0, \gamma_1, \gamma_2) \in \Gamma^3, \exists v \in V, \begin{cases} (\gamma_0, \gamma_1) \in \pi_1 \wedge v \in Ena(\gamma_0, \pi_1) \wedge Act(\gamma_0, \gamma_1) = \{v\} \\ (\gamma_1, \gamma_2) \in \pi_1 \wedge v \in Ena(\gamma_1, \pi_1) \wedge v \notin Act(\gamma_1, \gamma_2) \\ (\gamma_2, \gamma_1) \in \pi_1 \wedge v \in Ena(\gamma_2, \pi_1) \wedge v \notin Act(\gamma_2, \gamma_1) \\ Act(\gamma_1, \gamma_2) = Act(\gamma_2, \gamma_1) \end{cases}$$

Let σ be an execution of $d(\pi_1)$ starting from γ_2. Now, we define the following set of executions of π_1 (where the product operator denotes the concatenation of portions of executions):

$$\forall k' \in \mathbb{N}, \sigma_{k'} = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2).\left[(\gamma_2, \gamma_1)(\gamma_1, \gamma_2)\right]^{k'}.e$$

Note that, for any $k' \in \mathbb{N}$, the portion of execution $(\gamma_0, \gamma_1)(\gamma_1, \gamma_2).\left[(\gamma_2, \gamma_1)(\gamma_1, \gamma_2)\right]^{k'}$ is 1-bounded. Hence, any execution of $\{\sigma_{k'} | k' \in \mathbb{N}\}$ is k-bounded.

We can define a daemon d' in the following way:

$$\begin{cases} \forall \pi \in \Pi \setminus \{\pi_1\}, d'(\pi) = d(\pi) \\ d'(\pi_1) = d(\pi_1) \cup \{\sigma_{k'} | k' \in \mathbb{N}\} \end{cases}$$

Then, we can observe that $d' \in k\text{-}\mathcal{B}$ by construction and that, for any $k' \in \mathbb{N}$, the execution $\sigma_{k'} \in d'(\pi_1)$ does not satisfy the definition of k'-enabledness. Consequently, we prove that: $d' \in k\text{-}\mathcal{B} \setminus \bigcup_{k' \in \mathbb{N}} k'\text{-}\mathcal{E} = k\text{-}\mathcal{B} \setminus \mathcal{E}$. ■

3.2 Comparing Daemons

The four main characteristics presented in Section 3.1 provide a convenient way to define a particular class of daemons: this class simply combines the four characteristic properties. A formal definition follows.

Definition 3.7 (*Daemon class*)

Given a communication graph g and four sets of daemons

$$\begin{cases} C & \in & \{k\text{-}\mathcal{C} | k \in \{0, \ldots, diam(g)\}\} \\ B & \in & \{\mathcal{D}, k\text{-}\mathcal{B} | k \in \mathbb{N}^*\} \\ E & \in & \{\mathcal{D}, k\text{-}\mathcal{E} | k \in \mathbb{N}\} \\ F & \in & \{\mathcal{D}, \mathcal{WF}, \mathcal{SF}, \mathcal{GF}\} \end{cases},$$

the class of daemons $\mathcal{D}(C, B, E, F)$ is defined by $\mathcal{D}(C, B, E, F) = C \cap B \cap E \cap F$.

3.2.1 Comparing daemon classes

Now, each particular daemon instance d may belongs to several classes (those which include all possible executions under d). It is convenient to refer to the minimal class of d as the set of characteristics that strictly define d. A formal definition follows.

Definition 3.8 (*Minimal class*)

Given a communication graph g and a daemon d, the minimal class of d is the class of daemons $\mathcal{D}(C, B, E, F)$ such that:

$$\begin{cases} d \in \mathcal{D}(C, B, E, F) \\ \forall \mathcal{D}(C', B', E', F') \subsetneq \mathcal{D}(C, B, E, F), d \notin \mathcal{D}(C', B', E', F') \end{cases}$$

In any particular class, the canonical daemon of this class is a representative element of that class such that for any daemon d in the class, any execution allowed by d is also allowed by the canonical daemon. Simply put, the canonical daemon of a class is the largest element of this class with respect to allowed executions. A formal definition follows.

Definition 3.9 (*Canonical Daemon*)

For a given communication graph g and a class of daemons $\mathcal{D}(C, B, E, F)$, the canonical daemon $d(C, B, E, F)$ of $\mathcal{D}(C, B, E, F)$ is the daemon defined by:

$$\begin{cases} d(C, B, E, F) \in \mathcal{D}(C, B, E, F) \\ \forall d \in \mathcal{D}(C, B, E, F), \forall \pi \in \Pi, \forall \sigma \in \Sigma_\pi, \sigma \in d(\pi) \Rightarrow \sigma \in d(C, B, E, F)(\pi) \end{cases}$$

This way of viewing daemons as a set of possible executions (for a particular communication graph g) drives a natural "more powerful" relation definition. For a particular communication graph g, a daemon d is more powerful than another daemon d' if all executions allowed by d' are also allowed by d. Overall, d has more scheduling choices than d'. A formal definition follows.

Definition 3.10 (*More powerful relation*)

For a given communication graph g, we define the following binary relation $\preccurlyeq$ on $\mathcal{D}$:

$$\forall (d, d') \in \mathcal{D}, d \preccurlyeq d' \Leftrightarrow (\forall \pi \in \Pi, d(\pi) \subseteq d'(\pi))$$

If two daemons d and d' satisfy $d \preccurlyeq d'$, we say that d' is more powerful than d.

As with set inclusions, this "more powerful" relation induces a partial order, which is formally presented in the sequel.

Proposition 3.7

For any communication graph g, the binary relation $\preccurlyeq$ is a partial order on $\mathcal{D}$.

Proof: Let g be a communication graph. We are going to prove that the binary relation $\preccurlyeq$ is reflexive, antisymmetric and transitive. Then we show that this order is not total (*i.e.* that there exists some incomparable elements by $\preccurlyeq$ in $\mathcal{D}$).

For any daemon $d \in \mathcal{D}$, we have $\forall \pi \in \Pi, d(\pi) \subseteq d(\pi)$, that proves that $\forall d \in \mathcal{D}, d \preccurlyeq d$ (reflexivity of the binary relation $\preccurlyeq$).

Let d and d' be two daemons such that $d \preccurlyeq d'$ and $d' \preccurlyeq d$. Then, by definition, we have:

$$\left.\begin{array}{c} \forall \pi \in \Pi, d(\pi) \subseteq d'(\pi) \\ \forall \pi \in \Pi, d'(\pi) \subseteq d(\pi) \end{array}\right\} \Rightarrow \forall \pi \in \Pi, d(\pi) = d'(\pi)$$

In other words, we have $d = d'$ (antisymmetry of the binary relation $\preccurlyeq$).

Let d, d' and d'' be three daemons such that $d \preccurlyeq d'$ and $d' \preccurlyeq d''$. Then, by definition, we have:

$$\left.\begin{array}{c} \forall \pi \in \Pi, d(\pi) \subseteq d'(\pi) \\ \forall \pi \in \Pi, d'(\pi) \subseteq d''(\pi) \end{array}\right\} \Rightarrow \forall \pi \in \Pi, d(\pi) \subseteq d''(\pi)$$

In other words, we have $d \preccurlyeq d''$ (transitivity of the binary relation $\preccurlyeq$).

Let d be a daemon, π_1 and π_2 be two distributed protocols and σ_1 and σ_2 be two executions such that:

$$\left\{\begin{array}{c} \pi_1 \neq \pi_2 \\ \sigma_1 \notin d(\pi_1) \\ \sigma_2 \notin d(\pi_2) \end{array}\right.$$

Then, we can construct two daemons d_1 and d_2 in the following way:

$$\left\{\begin{array}{c} \forall \pi \in \Pi \setminus \{\pi_1\}, d_1(\pi) = d(\pi) \\ d_1(\pi_1) = d(\pi_1) \cup \{\sigma_1\} \end{array}\right. \text{, and } \left\{\begin{array}{c} \forall \pi \in \Pi \setminus \{\pi_2\}, d_2(\pi) = d(\pi) \\ d_2(\pi_2) = d(\pi_2) \cup \{\sigma_2\} \end{array}\right.$$

Then, we can deduce that $d_2(\pi_1) \subsetneq d_1(\pi_1)$ and $d_1(\pi_2) \subsetneq d_2(\pi_2)$, that proves that d_1 and d_2 are not comparable using the binary relation $\preccurlyeq$. ■

Another natural intuition is that if d is more powerful than d' and d belong to a particular daemon class, then d' also belongs to this class. This is formally demonstrated in the following.

Proposition 3.8

For a given communication graph g, for any daemons d and d' and for any class of daemons $\mathcal{D}(C, B, E, F)$, we have:

$$\left.\begin{array}{c} d' \preccurlyeq d \\ d \in \mathcal{D}(C, B, E, F) \end{array}\right\} \Rightarrow d' \in \mathcal{D}(C, B, E, F)$$

Proof: Let g be a communication graph, d and d' be two daemons and $\mathcal{D}(C, B, E, F)$ be a class of daemons such that: $d' \preccurlyeq d$ and $d \in \mathcal{D}(C, B, E, F)$.

Assume that $C = k\text{-}\mathcal{C}$ with $k \in \{0, \dots, diam(g)\}$. As $d \in \mathcal{D}(C, B, E, F) = C \cap B \cap E \cap F$, we know that $d \in k\text{-}\mathcal{C}$. By definition, we have:

$$\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \dots \in d(\pi), \forall i \in \mathbb{N}, \forall (u, v) \in V^2,$$
$$[u \neq v \wedge u \in Act(\gamma_i, \gamma_{i+1}) \wedge v \in Act(\gamma_i, \gamma_{i+1})] \Rightarrow dist(g, u, v) > k$$

As $d' \preccurlyeq d$, we know by definition that: $\forall \pi \in \Pi, d'(\pi) \subseteq d(\pi)$. Then, we obtain:

$$\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in d'(\pi) \subseteq d(\pi), \forall i \in \mathbb{N}, \forall (u,v) \in V^2, \\ [u \neq v \wedge u \in Act(\gamma_i, \gamma_{i+1}) \wedge v \in Act(\gamma_i, \gamma_{i+1})] \Rightarrow dist(g,u,v) > k$$

This implies that $d' \in k\text{-}\mathcal{C} = C$. We can prove in a similar way that $d' \in B$, $d' \in E$ and $d' \in F$. Consequently, we obtain that $d' \in C \cap B \cap E \cap F = \mathcal{D}(C,B,E,F)$, that proves the result. ■

3.2.2 Preserving execution properties

Meaningful distributed protocols provide non-trivial properties when operated. A property can be defined as a predicate on executions, valued with *true* when the predicate is satisfied and *false* otherwise. A distributed protocol satisfies a property if its every executions satisfy the corresponding predicate. Conversely, a property is impossible to satisfy if no distributed protocol is such that any of its executions satisfies the corresponding predicate. Formal definitions follow.

Definition 3.11 (*Execution property*)

For a given communication graph g, a property of execution p is a function that associates to each execution a Boolean value.

$$\begin{array}{rcrcl} p & : & \Sigma_\Pi & \longrightarrow & \{true, false\} \\ & & \sigma & \longmapsto & p(\sigma) \in \{true, false\} \end{array}$$

Definition 3.12 (*Property satisfaction*)

For a given communication graph g, a distributed protocol π satisfies a property of execution p under a daemon d (denoted by $\pi \overset{d}{\models} p$) if and only if $\forall \sigma \in d(\pi), p(\sigma) = true$.

Definition 3.13 (*Property impossibility*)

For a given communication graph g, it is impossible to satisfy a property of execution p under a daemon d (denoted by $d \not\models p$) if and only if $\forall \pi \in \Pi, \exists \sigma \in d(\pi), p(\sigma) = false$.

The "more powerful" meaning that is associated to the $\preccurlyeq$ relation permits to intuitively understand the two following theorems. If a property is guaranteed by a distributed protocol under a daemon d, it is also guaranteed using the same distributed protocol under any "less powerful" daemon d' (the executions allowed by d' are a – possibly strict – subset of those allowed by d). Similarly, if a property cannot be guaranteed by any distributed protocol under a daemon d, it is also impossible to guarantee this property under a "more powerful" daemon d' (the executions that falsifies the property in those allowed by d are also present in those allowed by d'). A formal treatment follows.

Theorem 3.1

For a given communication graph g, let p be a property of execution satisfied by a distributed protocol π under a daemon d. Then,

$$\forall d' \in \mathcal{D}, d' \preccurlyeq d \Rightarrow \pi \overset{d'}{\models} p$$

Proof: Let g be a communication graph, p be a property of execution satisfied by a distributed protocol π_1 under a daemon d. By definition, we have:

$$\forall \sigma \in d(\pi_1), p(\sigma) = true$$

Assume now that d' is a daemon such that $d' \preccurlyeq d$. By definition, we have:

$$\forall \pi \in \Pi, d'(\pi) \subseteq d(\pi)$$

Consequently, we have:

$$\forall \sigma \in \Sigma_{\pi_1}, \sigma \in d'(\pi_1) \Rightarrow \sigma \in d(\pi_1) \Rightarrow p(\sigma) = true$$

By definition, we obtain that: $\pi_1 \overset{d'}{\models} p$, that proves the theorem. ■

Theorem 3.2

For a given communication graph g, let p be a property of execution impossible under a daemon d. Then,

$$\forall d' \in \mathcal{D}, d \preccurlyeq d' \Rightarrow d' \not\models p$$

Proof: Let g be a communication graph, p be a property of execution impossible under a daemon d. By definition, we have:

$$\forall \pi \in \Pi, \exists \sigma \in d(\pi), p(\sigma) = false$$

Assume now that d' is a daemon such that $d \preccurlyeq d'$. By definition, we have:

$$\forall \pi \in \Pi, d(\pi) \subseteq d'(\pi)$$

Consequently, we have:

$$\forall \pi \in \Pi, \exists \sigma \in d(\pi) \subseteq d'(\pi), p(\sigma) = false$$

By definition, we obtain that: $d' \not\models p$, that proves the theorem. ■

A less obvious result shows that dealing with canonical daemons (rather than with the classes they represent) is sufficient for comparison purposes. The two derived corollaries demonstrate that using characteristic daemons is also valid for proving properties (or lack hereof) executions. This is formalized in the sequel.

Theorem 3.3

For a given communication graph g, let $d(C, B, E, F)$ and $d(C', B', E', F')$ be two canonical daemons. Then,

$$d(C, B, E, F) \preccurlyeq d(C', B', E', F') \Leftrightarrow \begin{cases} C \subseteq C' \\ B \subseteq B' \\ E \subseteq E' \\ F \subseteq F' \end{cases}$$

Proof: We first prove the "$\Leftarrow$" part of the theorem.

Assume that we have a communication graph g and two canonical daemons $d(C, B, E, F)$ and $d(C', B', E', F')$ such that:

$$\begin{cases} C \subseteq C' \\ B \subseteq B' \\ E \subseteq E' \\ F \subseteq F' \end{cases}$$

We can deduce that $C \cap B \cap E \cap F \subseteq C' \cap B' \cap E' \cap F'$. Then, by the definition of a class of daemons, we have:

$$\mathcal{D}(C, B, E, F) \subseteq \mathcal{D}(C', B', E', F')$$

By the definition of a canonical daemon, we know that:

$$d(C, B, E, F) \in \mathcal{D}(C, B, E, F)$$

Hence, we have:

$$d(C, B, E, F) \in \mathcal{D}(C', B', E', F')$$

As $d(C', B', E', F')$ is the canonical daemon of the class $\mathcal{D}(C', B', E', F')$, we know by definition that:

$$\forall \pi \in \Pi, \forall \sigma \in \Sigma_\pi, \sigma \in d(C, B, E, F)(\pi) \Rightarrow \sigma \in d(C', B', E', F')(\pi)$$

In other words,

$$\forall \pi \in \Pi, d(C, B, E, F)(\pi) \subseteq d(C', B', E', F')(\pi)$$

This means that: $d(C, B, E, F) \preccurlyeq d(C', B', E', F')$, that ends the first part of the proof.

Then, we prove the "$\Rightarrow$" part of the theorem.

Assume that we have a communication graph g and two canonical daemons $d(C, B, E, F)$ and $d(C', B', E', F')$ such that: $d(C, B, E, F) \preccurlyeq d(C', B', E', F')$.

By definition of the $\preccurlyeq$ relation, we know that:

$$\forall \pi \in \Pi, d(C, B, E, F)(\pi) \subseteq d(C', B', E', F')(\pi)$$

Let d be a daemon of $\mathcal{D}(C, B, E, F)$. As $d(C, B, E, F)$ is the canonical daemon

of the class of daemons $\mathcal{D}(C,B,E,F)$, we know that:

$$\begin{aligned}\forall \pi \in \Pi, \forall \sigma \in \Sigma_\pi, \sigma \in d(\pi) &\Rightarrow \sigma \in d(C,B,E,F)(\pi)\\ &\Rightarrow \sigma \in d(C',B',E',F')(\pi)\end{aligned}$$

In other words, we have: $\forall \pi \in \Pi, d(\pi) \subseteq d(C',B',E',F')(\pi)$. By the definition of the $\preccurlyeq$ relation, that implies that:

$$\forall d \in \mathcal{D}(C,B,E,F), d \preccurlyeq d(C',B',E',F')$$

As $d(C',B',E',F')$ is the canonical daemon of the class of daemons $\mathcal{D}(C',B',E',F')$, we know that $d(C',B',E',F') \in \mathcal{D}(C',B',E',F')$ and Proposition 3.8 allow us to state that:

$$\forall d \in \mathcal{D}(C,B,E,F), d \in \mathcal{D}(C',B',E',F')$$

In other words, $C \cap B \cap E \cap F = \mathcal{D}(C,B,E,F) \subseteq \mathcal{D}(C',B',E',F') = C' \cap B' \cap E' \cap F'$.

Assume by contradiction that $C' \subsetneq C$. By the properties of boundedness, enabledness and fairness (see propositions of Section 3), we know that $(C \setminus C') \cap B \cap E \cap F \neq \emptyset$. In this way, we know that there exists a daemon d such that $d \in C \cap B \cap E \cap F$ and $d \notin C'$. Then, we can deduce that $d \notin C' \cap B' \cap E' \cap F'$, that contradicts $C \cap B \cap E \cap F \subseteq C' \cap B' \cap E' \cap F'$.

By the same way, we can prove that:

$$\begin{cases} C \subseteq C' \\ B \subseteq B' \\ E \subseteq E' \\ F \subseteq F' \end{cases}$$

This result ends the proof. ■

Corollary 3.1

For a given communication graph g, let $d(C,B,E,F)$ and $d(C',B',E',F')$ be two canonical daemons. Then, for any property of execution p satisfied by a distributed protocol π under $d(C,B,E,F)$, we have:

$$\left.\begin{array}{l} C' \subseteq C \\ B' \subseteq B \\ E' \subseteq E \\ F' \subseteq F \end{array}\right\} \Rightarrow \pi \overset{d(C',B',E',F')}{\models} p$$

Proof: This result is a direct corollary from Theorems 1 and 3. ■

Corollary 3.2

For a given communication graph g, let $d(C,B,E,F)$ and $d(C',B',E',F')$ be two canonical daemons. Then, for any property of execution p impossible under

$d(C,B,E,F)$, we have:

$$\left.\begin{array}{l} C \subseteq C' \\ B \subseteq B' \\ E \subseteq E' \\ F \subseteq F' \end{array}\right\} \Rightarrow d(C',B',E',F') \not\models p$$

Proof: This result is a direct corollary from Theorems 2 and 3. ■

3.2.3 The Case of the Synchronous Daemon

Although we did not describe it in the previous sections, the synchronous daemon plays a very important part in the self-stabilization literature. First introduced by Herman [Her90] to enable analytical tractability of probabilistic self-stabilizing distributed protocols, it was later used in a number of works, either to demonstrate impossibility results (due to initial symmetry [PBT00]) or to enable efficient solution to existing problems (due to the single execution generated [DHT04]). A synchronous daemon simply executes every enabled vertex at every action. A formal definition follows.

Definition 3.14

Given a communication graph g, the synchronous daemon (denoted by sd) is defined by:

$$\forall \pi \in \Pi, \forall \sigma = (\gamma_0,\gamma_1)(\gamma_1,\gamma_2)\ldots \in sd(\pi), \forall i \in \mathbb{N}, \forall v \in V, v \in Ena(\gamma_i,\pi) \Rightarrow v \in Act((\gamma_i,\gamma_{i+1}))$$

We first show that there is a connection between enabledness and synchrony. Indeed the synchronous daemon cannot prevent an enabled vertex from being activated, even for a single action.

Proposition 3.9

For any given communication graph g, we have: $0\text{-}\mathcal{E} = \{sd\}$.

Proof: Let g be a communication graph and d be a daemon such that $d \in 0\text{-}\mathcal{E}$. We are going to prove that $d = sd$. By definition, we have:

$$\begin{array}{l} \exists k \in \mathbb{N}, \forall \pi \in \Pi, \forall \sigma = (\gamma_0,\gamma_1)(\gamma_1,\gamma_2)\ldots \in d(\pi), \forall (i,j) \in \mathbb{N}^2, \forall v \in V, \\ \quad \big[[v \in Act(\gamma_i,\gamma_{i+1}) \wedge (\forall l \in \mathbb{N}, l < i \Rightarrow v \notin Act(\gamma_l,\gamma_{l+1}))] \\ \quad \Rightarrow |\{l \in \mathbb{N} | l < i \wedge v \in Ena(\gamma_l,\pi)\}| = 0\big] \wedge \\ \quad \big[[i < j \wedge v \in Act(\gamma_i,\gamma_{i+1}) \wedge v \in Act(\gamma_j,\gamma_{j+1}) \wedge \\ \quad (\forall l \in \mathbb{N}, i < l < j \Rightarrow v \notin Act(\gamma_l,\gamma_{l+1}))] \\ \quad \Rightarrow |\{l \in \mathbb{N} | i < l < j \wedge v \in Ena(\gamma_l,\pi)\}| = 0\big] \wedge \\ \quad \big[[v \in Act(\gamma_i,\gamma_{i+1}) \wedge (\forall l \in \mathbb{N}, l > i \Rightarrow v \notin Act(\gamma_l,\gamma_{l+1}))] \\ \quad \Rightarrow |\{l \in \mathbb{N} | l > i \wedge v \in Ena(\gamma_l,\pi)\}| = 0\big] \end{array}$$

In other words, no action (γ, γ') of any execution of $d(\pi)$ for any distributed protocol π can satisfy: $\exists v \in V, v \in \mathit{Ena}(\gamma, \pi) \wedge v \notin \mathit{Act}(\gamma, \gamma')$. Hence:

$$\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in d(\pi), \forall i \in \mathbb{N}, \forall v \in V, \\ v \notin \mathit{Ena}(\gamma_i, \pi) \vee v \in \mathit{Act}(\gamma_i, \gamma_{i+1})$$

As $v \in \mathit{Act}(\gamma_i, \gamma_{i+1})$ implies that $v \in \mathit{Ena}(\gamma_i, \pi)$, this property is equivalent to the following:

$$\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in d(\pi), \forall i \in \mathbb{N}, \forall v \in V, \\ v \in \mathit{Ena}(\gamma_i, \pi) \Rightarrow v \in \mathit{Act}(\gamma_i, \gamma_{i+1})$$

By the definition of the synchronous daemon, this means that $d = sd$, that ends the proof. ■

It may first come to a surprise that boundedness is absolutely not related to synchrony, but as we pointed out previously, boundedness is also not related to fairness. The exact characteristics of the synchronous daemon are captured by the following proposition.

Proposition 3.10

Given a communication graph g, $\mathcal{D}(0\text{-}\mathcal{C}, \mathcal{D}, 0\text{-}\mathcal{E}, \mathcal{SF})$ is the minimal class of sd. Moreover, $sd = d(0\text{-}\mathcal{C}, \mathcal{D}, 0\text{-}\mathcal{E}, \mathcal{SF})$.

Proof: In a first time, we prove that $sd \in 0\text{-}\mathcal{C} \setminus 1\text{-}\mathcal{C}$. It is obvious that $sd \in 0\text{-}\mathcal{C} = \mathcal{D}$. By contradiction, assume that $sd \in 1\text{-}\mathcal{C}$. Let $\pi \in \Pi$ be a distributed protocol such that:

$$\exists (\gamma, \gamma') \in \pi, \mathit{Ena}(\gamma, \pi) = V$$

Then, by definition of the synchronous daemon, the first action of any execution $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in sd(\pi)$ starting from $\gamma_0 = \gamma$ satisfies: $\mathit{Act}(\gamma_0, \gamma_1) = V$. Consequently, σ does not satisfy the property of executions allowed by a 1-central daemon, that contradicts $sd \in 1\text{-}\mathcal{C}$ and proves the result.

In a second time, we prove that $sd \in \bar{\mathcal{B}}$. As it is obvious that $sd \in \mathcal{D}$, assume by contradiction that there exists $k \in \mathbb{N}^*$ such that $sd \in k\text{-}\mathcal{B}$. Then, consider a distributed protocol π such that:

$$\exists (v, u) \in V^2, \exists (\gamma_0, \ldots, \gamma_{k+3}) \in \Gamma^{k+4}, \begin{cases} (\gamma_0, \gamma_1) \in \pi \wedge \mathit{Ena}(\gamma_0, \pi) = \{v\} \\ \forall i \in \{1, \ldots, k+1\}, (\gamma_i, \gamma_{i+1}) \in \pi \\ \forall i \in \{1, \ldots, k+1\}, \mathit{Ena}(\gamma_i, \pi) = \{u\} \\ (\gamma_{k+2}, \gamma_{k+3}) \in \pi \wedge \mathit{Ena}(\gamma_{k+2}, \pi) = \{v\} \\ \mathit{Ena}(\gamma_{k+3}, \pi) = \emptyset \end{cases}$$

We can observe that the execution σ defined by $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots (\gamma_{k+2}, \gamma_{k+3})$ satisfies $\sigma \in sd(\pi)$. But, on the other hand, we have:

$$\exists \pi \in \Pi, \exists \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in d(\pi), \exists (i = 0, j = k+2) \in \mathbb{N}^2, \exists v \in V, \\ [i < j \wedge v \in \mathit{Act}(\gamma_i, \gamma_{i+1}) \wedge v \in \mathit{Act}(\gamma_j, \gamma_{j+1}) \wedge \\ (\forall l \in \mathbb{N}, i < l < j \Rightarrow v \notin \mathit{Act}(\gamma_l, \gamma_{l+1}))] \\ \wedge \exists u \in V \setminus \{v\}, |\{l \in \mathbb{N} | i \leq l < j \wedge u \in \mathit{Act}(\gamma_l, \gamma_{l+1})\}| = k+1$$

By the definition of a k-bounded daemon, this implies that $sd \notin k\text{-}\mathcal{B}$.

In a third time, we prove that $sd \in 0\text{-}\mathcal{E}$. By Proposition 8, we know that $0\text{-}\mathcal{E} = \{sd\}$. This implies that $sd \in 0\text{-}\mathcal{E}$.

In a fourth time, we prove that $sd \in \mathcal{SF} \setminus \mathcal{GF}$. We start by proving that $sd \in \mathcal{SF}$. By the definition of the synchronous daemon, we know that:

$$\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in \Sigma_\pi, \forall v \in V, (\exists i \in \mathbb{N}, v \in Ena(\gamma_i, \pi)) \\ \Rightarrow v \in Act(\gamma_j, \gamma_{j+1})$$

Consequently, we have:

$$\forall \pi \in \Pi, \forall \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in \Sigma_\pi, \\ [\exists i \in \mathbb{N}, \exists v \in V, (\forall j \geq i, \exists k \geq j, v \in Ena(\gamma_k, \pi)) \wedge (\forall j \geq i, v \notin Act(\gamma_j, \gamma_{j+1}))] \\ \Rightarrow \sigma \notin sd(\pi)$$

By the definition of a strongly fair daemon, this implies that $sd \in \mathcal{SF}$. Now, we prove that $sd \notin \mathcal{GF}$. Consider a distributed protocol π such that:

$$\exists (\gamma, \gamma', \gamma'') \in \Gamma^3, \begin{cases} (\gamma, \gamma') \in \pi \wedge Act(\gamma, \gamma') \subsetneq Ena(\gamma, \pi) \\ (\gamma, \gamma'') \in \pi \wedge Act(\gamma, \gamma'') = Ena(\gamma, \pi) \\ (\gamma'', \gamma) \in \pi \wedge Act(\gamma'', \gamma) = Ena(\gamma'', \pi) \end{cases}$$

We can construct an execution σ of π starting from γ in the following way: $\sigma = (\gamma, \gamma'')(\gamma'', \gamma)(\gamma, \gamma'') \ldots$. We can observe that $\sigma \in sd(\pi)$ (since at each action, any enabled vertex is activated). Consequently, we have:

$$\exists \pi \in \Pi, \exists \sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2) \ldots \in sd(\pi), \exists (\gamma, \gamma') \in \pi, \\ \exists i = 0 \in \mathbb{N}, (\forall j \geq i, \exists k = 2j \geq j, \gamma_k = \gamma) \wedge (\forall j \geq i, (\gamma_j, \gamma_{j+1}) \neq (\gamma, \gamma'))$$

By the definition of a Gouda fair daemon, this implies that $sd \notin \mathcal{GF}$.

The four previous results imply that $\mathcal{D}(0\text{-}\mathcal{C}, \mathcal{D}, 0\text{-}\mathcal{E}, \mathcal{SF})$ is the minimal class of sd. As $\mathcal{D}(0\text{-}\mathcal{C}, \mathcal{D}, 0\text{-}\mathcal{E}, \mathcal{SF}) \subseteq 0\text{-}\mathcal{E}$ by definition and $0\text{-}\mathcal{E} = \{sd\}$ by Proposition 3.9, we can deduce that $\mathcal{D}(0\text{-}\mathcal{C}, \mathcal{D}, 0\text{-}\mathcal{E}, \mathcal{SF}) = \{sd\}$. Then, the definition of a canonical daemon implies that $sd = d(0\text{-}\mathcal{C}, \mathcal{D}, 0\text{-}\mathcal{E}, \mathcal{SF})$, that ends the proof. ■

3.2.4 A map of classical daemons

We are now ready to present our map for "classical" daemons (*i.e.* daemons most frequently used in the literature). Using our taxonomy, these daemons can be defined as follows.

Definition 3.15 (*Classical daemons*)

Given a communication graph g, the classical daemons of the literature are defined as follows:

- *The unfair daemon (denoted by ufd) is $d(\mathcal{D}, \mathcal{D}, \mathcal{D}, \mathcal{D})$.*
- *The weakly fair daemon (denoted by wfd) is $d(\mathcal{D}, \mathcal{D}, \mathcal{D}, \mathcal{WF})$.*
- *The strongly fair daemon (denoted by sfd) is $d(\mathcal{D}, \mathcal{D}, \mathcal{D}, \mathcal{SF})$.*
- *The Gouda fair daemon (denoted by gfd) is $d(\mathcal{D}, \mathcal{D}, \mathcal{D}, \mathcal{GF})$.*

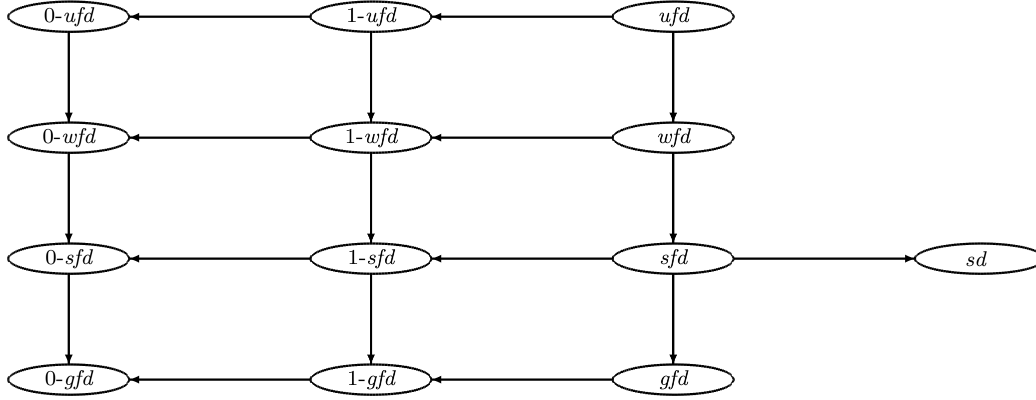

Figure 3.7: Relationship between classical daemons (an arrow from a daemon d to a daemon d' means that $d' \preccurlyeq d$, note that we remove all arrows obtained by transitivity).

- *The locally central unfair daemon (denoted by* 1-*ufd) is* $d(1\text{-}\mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{D})$.
- *The locally central weakly fair daemon (denoted by* 1-*wfd) is* $d(1\text{-}\mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{WF})$.
- *The locally central strongly fair daemon (denoted by* 1-*sfd) is* $d(1\text{-}\mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{SF})$.
- *The locally central Gouda fair daemon (denoted by* 1-*gfd) is* $d(1\text{-}\mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{GF})$.
- *The central unfair daemon (denoted by* 0-*ufd) is* $d(0\text{-}\mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{D})$.
- *The central weakly fair daemon (denoted by* 0-*wfd) is* $d(0\text{-}\mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{WF})$.
- *The central strongly fair daemon (denoted by* 0-*sfd) is* $d(0\text{-}\mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{SF})$.
- *The central Gouda fair daemon (denoted by* 0-*gfd) is* $d(0\text{-}\mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{GF})$.

Now, our main theorem (Theorem 3.3) permits to map the relationships between all classical daemons in the literature is a rather compact format. For any given communication graph g, Figure 3.7 depicts graphically those relationships.

3.3 Daemon Transformers

As it is easier to write distributed protocols under daemons providing strong properties (that is, under weak daemons that allow only a limited set of possible executions, such as a central or a bounded daemon), many authors provide distributed protocols to simulate the operation of a weak daemon under a strong one. Such distributed protocols are called daemon transformers. Note that several works in the area of self-stabilization may be used as daemon transformers although they were not initially designed with this goal in mind (*e.g.* a self-stabilizing token circulation distributed protocol that performs under the unfair distributed daemon can easily be turned into a daemon transformer that provides a central daemon out of an unfair distributed one).

In the following, we propose a survey of the main daemon transformers that

also preserve the property of self-stabilization. That is, the distributed protocol transforming the daemon is a self-stabilizing one. Figure 3.8 summarizes this survey and maps for each daemon transformer the initial daemon and the simulated one. We restrict ourselves to deterministic daemon transformers in order to be able to exactly compute the characteristics of the simulated daemon. Note that features of the emulated daemon (centrality, fairness, boundedness, and enabledness) provided in the following are satisfied only after the stabilization of the daemon transformer. In the sequel, we use the notation $d \longmapsto d'$ to denote that a daemon transformer simulates d' while operating under d.

Alternator-based daemon transformers. In 1997, Gouda and Haddix [GH97] introduced the alternator problem. Roughly speaking, the aim is to design a distributed protocol such that no neighbors are enabled simultaneously yet ensures that some fairness property holds (namely, between any two actions of a particular vertex, any of its neighbors may execute at most one action). They claim that this distributed protocol is useful to simulate a locally central daemon under a distributed one. Actually, this distributed protocol ensures the following daemon transformation: $\mathit{ufd} \longmapsto d(1\text{-}\mathcal{C}, \mathcal{WF}, n^2\text{-}\mathcal{B}, n^2\text{-}\mathcal{E})$ and works only on communication graphs reduced to chains. Johnen *et al.* [JADT02] later designed an alternator for any oriented tree but require the initial daemon being weakly fair. In other words, they provide the following daemon transformer: $\mathit{wfd} \longmapsto d(1\text{-}\mathcal{C}, \mathcal{WF}, n^2\text{-}\mathcal{B}, n^2\text{-}\mathcal{E})$. Finally, Gouda and Haddix [GH07] provided an alternator for an arbitrary underlying communication graph that provides the following daemon transformation: $\mathit{wfd} \longmapsto d(1\text{-}\mathcal{C}, \mathcal{WF}, \mathit{diam}(g)\text{-}\mathcal{B}, (n \times \mathit{diam}(g))\text{-}\mathcal{E})$. This last transformer makes the following assumption: the graph is identified (that is, every vertex has a unique identifier) and each vertex knows the cyclic distance of the graph (the cyclic distance is defined as the number of edges of the longest simple cycle if the graph has cycles, and two otherwise).

Mutual exclusion-based daemon transformers. The classical mutual exclusion problem requires that no two vertices are simultaneously in critical section and that every vertex infinitely often enters critical section. In this way, any self-stabilizing mutual exclusion distributed protocol may be turned into a daemon transformer that provides a central weakly fair daemon. In his seminal work on self-stabilization [Dij74], Dijkstra proposed a self-stabilizing mutual exclusion distributed protocol for ring topologies (using a token circulation) under a distributed unfair daemon. His distributed protocol needs however a distinguished vertex (that is, one vertex executes a protocol that is different from every other). Formally, we can derive the following daemon transformation from this distributed protocol: $\mathit{ufd} \longmapsto d(\mathit{diam}(g)\text{-}\mathcal{C}, \mathcal{WF}, 1\text{-}\mathcal{B}, n\text{-}\mathcal{E})$. From this first distributed protocol, several works later revisited the mutual exclusion problem. From a daemon transformation viewpoint, the most interesting ones follow. Using a token circulation, Beauquier *et al.* ([BPBJDL02]) provide a $\mathit{ufd} \longmapsto d(\mathit{diam}(g)\text{-}\mathcal{C}, \mathcal{WF}, \mathit{deg}^+(g)\text{-}\mathcal{B}, n \times$

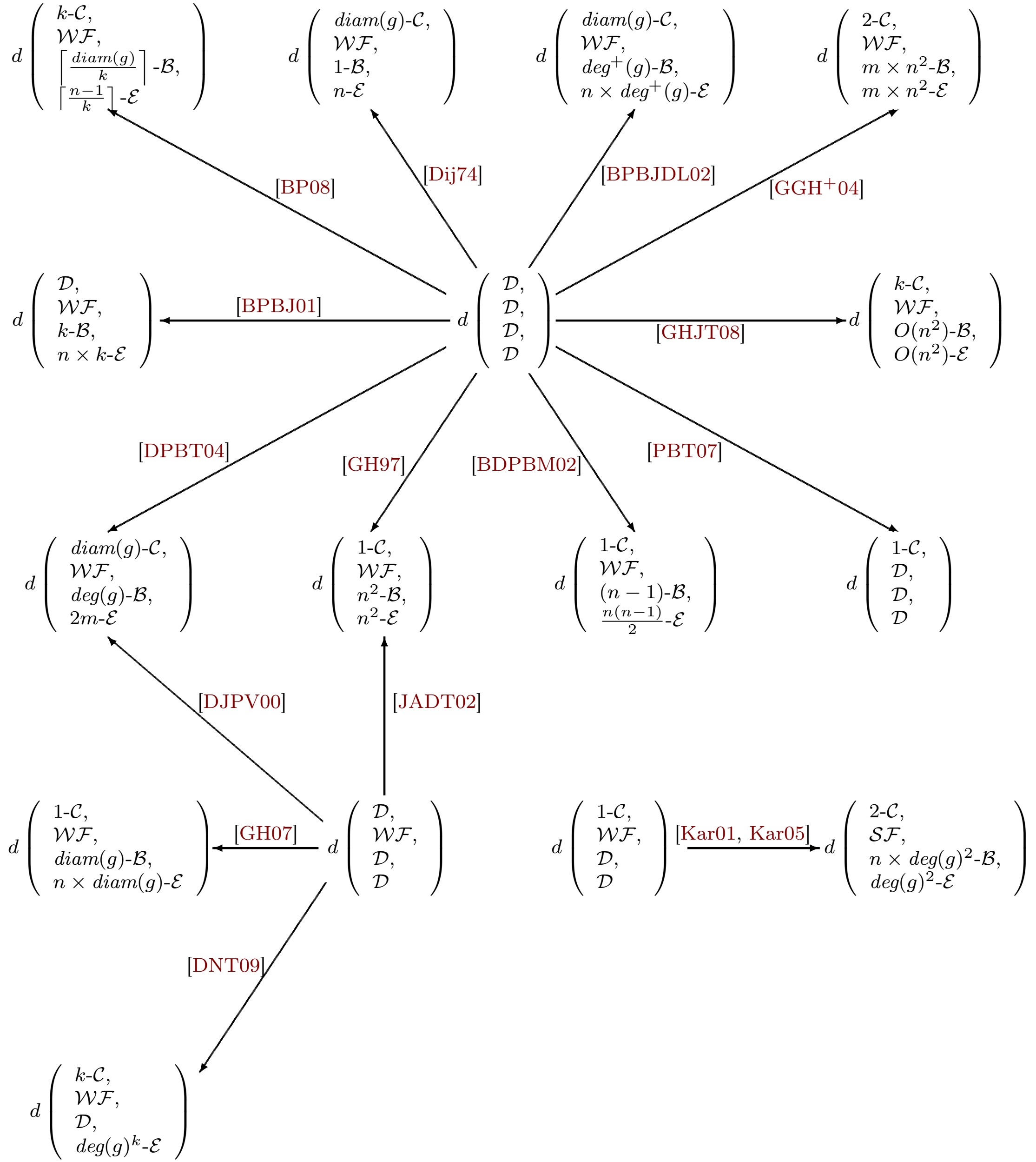

Figure 3.8: Summary of existing daemon transformers. An arrow from a daemon to another one means that the related work provides a transformer from the first to the second.

$deg^+(g)$-$\mathcal{E}$) daemon transformation on oriented communication graph whenever the communication graph is strongly connected. Still on communication graphs with a distinguished vertex, Datta *et al.* [DJPV00] provided a self-stabilizing depth-first token circulation that perform the following daemon transformation: $wfd \longmapsto d(diam(g)\text{-}\mathcal{C}, \mathcal{WF}, deg(g)\text{-}\mathcal{B}, 2m\text{-}\mathcal{E})$. Finally, Datta *et al.* [DPBT04] improved this result enabling the same daemon transformation but starting from an unfair daemon (more formally, they achieve the following daemon transformation: $ufd \longmapsto d(diam(g)\text{-}\mathcal{C}, \mathcal{WF}, deg(g)\text{-}\mathcal{B}, 2m\text{-}\mathcal{E})$) and they do not require the existence of a distinguished vertex.

Local mutual exclusion-based daemon transformers. Local mutual exclusion refines mutual exclusion since it requires the same exclusion and liveness properties but only within a vicinity around each vertex (and not for the whole communication graph as for the – global – mutual exclusion problem). In other words, a k-local mutual exclusion distributed protocol ensures that no two vertices are simultaneously in critical section if their distance is less than k and that any vertex enters infinitely often in critical section. Hence, we can easily design a daemon transformer providing a weakly fair k-central daemon from such a distributed protocol. Note that the aforementioned alternator distributed protocols solve a particular instance of 1-local mutual exclusion.

A classical solution to 1-local mutual exclusion has been proposed by Beauquier *et al.* [BDPBM02] using unbounded memory at each vertex. This distributed protocol ensures the following daemon transformation: $ufd \longmapsto d(1\text{-}\mathcal{C}, \mathcal{WF}, (n-1)\text{-}\mathcal{B}, \frac{n(n-1)}{2}\text{-}\mathcal{E})$. Using only a bounded memory, Gairing *et al.* provided [GGH⁺04] a 2-local mutual exclusion that can be turned into a $ufd \longmapsto d(2\text{-}\mathcal{C}, \mathcal{WF}, m \times n^2\text{-}\mathcal{B}, m \times n^2\text{-}\mathcal{E})$ daemon transformer.

Several works give more general solutions dealing with k-local mutual exclusion for any integer k. For example, Goddart *et al.* generalize [GHJT08] the work of Gairing *et al.* [GGH⁺04]. Their solution performs the $ufd \longmapsto d(k\text{-}\mathcal{C}, \mathcal{WF}, O(n^2)\text{-}\mathcal{B}, O(n^2)\text{-}\mathcal{E})$ daemon transformation. Using a local clock synchronization, Boulinier and Petit provide [BP08] a wavelets distributed protocol that can be used for k-local mutual exclusion. In this way, their distributed protocol gives the following daemon transformation: $ufd \longmapsto d(k\text{-}\mathcal{C}, \mathcal{WF}, \left\lceil \frac{diam(g)}{k} \right\rceil\text{-}\mathcal{B}, \left\lceil \frac{n-1}{k} \right\rceil\text{-}\mathcal{E})$. Danturi *et al.* [DNT09] deal with dining philosophers with generic conflicts under a distributed weakly fair daemon. The main idea is to clearly distinguish the communication graph from the conflict graph. If we consider that two vertices are in conflict if they are at distance less than k from each other, this protocol ensures k-local mutual exclusion. This distributed protocol provides the $wfd \longmapsto d(k\text{-}\mathcal{C}, \mathcal{WF}, \mathcal{D}, deg(g)^k\text{-}\mathcal{E})$ daemon transformation but requires each vertex to be the root of a tree spanning its k-neighborhood.

Finally, Potop-Butucaru and Tixeuil introduced in [PBT07] a weaker version of 1-local mutual exclusion by replacing the fairness property by a progress property. This new problem was called a conflict manager and leads to the $ufd \longmapsto 1\text{-}ufd$

daemon transformation. To our knowledge, this daemon transformer is the only one to perform a transformation according to a single identifier daemon characteristic.

Note that all solutions presented in this paragraph require the communication graph to be identified.

Other daemon transformers. Even if they transform several characteristics of daemons (with the notable exception of [PBT07]), all previously mentioned daemon transformers are designed for transforming only the distribution of daemons. Indeed, only a few works dealt with transforming other daemon characteristics.

Regarding fairness transformation, Karaata [Kar01] provided a daemon transformer to perform strong fairness under weak fairness. More formally, this distributed protocol is a $1\text{-}wfd \longmapsto d(2\text{-}\mathcal{C}, \mathcal{SF}, n \times deg(g)^2\text{-}\mathcal{B}, deg(g)^2\text{-}\mathcal{E})$ daemon transformer. This distributed protocol needs the graph to be identified and each vertex to have an unbounded memory. Karaata later refined [Kar05] the distributed protocol to perform exactly the same daemon transformation but requiring only an identified graph.

Using cross-over composition, Beauquier *et al.*[BPBJ01] gave a generic transformer for enabledness. More precisely, they design a $ufd \longmapsto d(\mathcal{D}, \mathcal{WF}, k\text{-}\mathcal{B}, n \times k\text{-}\mathcal{E})$ daemon transformer whenever a transformer that provides k-boundedness is available.

CHAPTER 4

Fault Tolerance

> Focus on remedies, not faults.
>
> Jack Nicklaus

Contents

4.1 Tolerating Transient Fault Patterns **59**
  4.1.1 Weakening Self-Stabilization 60
  4.1.2 Enhancing Self-Stabilization 61
4.2 Tolerating Composite Fault Patterns **62**
  4.2.1 Fault-Tolerant Self-Stabilization 62
  4.2.2 Byzantine Tolerant Self-Stabilization 63
  4.2.3 Strict Stabilization 64
4.3 Summary **66**

In Chapters 2 and 3, we presented in details our models of distributed systems and of faults. In particular, we defined a taxonomy of faults that can capture any incorrect behavior of any component of the distributed system using our concept of fault pattern (see Section 2.4.2). Now, we are able to present fault-tolerant distributed protocols. That is, distributed protocols that have the ability to still behave correctly (in some extent) in spite of the occurrence of faults. Obviously, it is impossible to provide a distributed protocol that can tolerate any fault pattern without any incidence on its behavior or its results. Therefore, numerous fault tolerance schemes have been proposed so far in distributed systems. Each of them is convenient to tolerate a given class of fault patterns with some properties and sometimes it is tricky to compare these fault tolerance schemes with each other.

According to [Tix09], we can classify fault tolerance schemes according to their masking property. A fault tolerance scheme is masking when an external observer of the distributed system cannot be aware of the occurrence of faults (in other words, a masking fault tolerant scheme hides the occurrence of faults to the external observer). In the other hand, a non-masking fault tolerance scheme does not fulfill this requirement: the external observer may see the effects of faults over a certain period of time.

Even if a masking solution may appear preferable at the first glance (since it corresponds to the intuition of fault tolerance), we must consider other aspects of the problem. Indeed, masking solutions are usually more costly (in time and in system resources), more difficult to design, and tolerated classes of fault patterns

is often more restricted. Hence, according to the application and to the context, it is sometimes preferable to use a non-masking fault tolerant scheme. For problems such as routing, where incorrect behavior for a short period of time does not have catastrophic consequences, it may be useful to design a light non-masking fault tolerant distributed protocol. But, at the contrary, one must choose a masking approach for a critical application (*e.g.* train traffic control application). Two major categories of fault tolerance schemes can be distinguished:

Robust distributed protocols: this class of distributed protocols guarantees that the distributed protocol works correctly in spite of the occurrence of faults. Note that such distributed protocols perform their properties only if a limited number of faults of a given nature strike the distributed system. Typically, robust distributed protocols are masking since correct elements of the distributed system still behave correctly until faults remain in the tolerated range. Usually, robust distributed protocols are designed to deal with permanent crash or Byzantine fault patterns. In the first case, we say that they are fault-tolerant, in the second they are Byzantine-tolerant.

Self-stabilizing distributed protocols: this class of distributed protocols relies on the hypothesis that the considered fault patterns are transient (with no other assumption on nature or span). A distributed protocol is self-stabilizing [Dij74] if it guarantees that, starting from any arbitrary configuration (that models the effects of transient faults on the distributed system), any execution eventually reaches a correct behavior. Typically, a self-stabilizing distributed protocol is non-masking since the distributed system has an erratic behavior between the end of transient faults and the stabilization of the system.

In this thesis, we focus mainly on the non-masking fault tolerance approach but we consider more severe fault model than the one tolerated by self-stabilization. Indeed, our goal is the design of distributed protocols able to handle any transient and permanent crash fault pattern or any transient and intermittent Byzantine fault pattern. Therefore, we focus on self-stabilizing distributed protocols that are moreover resilient to permanent or intermittent faults. In the following of this chapter, we survey fault tolerance schemes that allow to deal with such fault patterns.

First, Section 4.1 defines formally self-stabilization and surveys its main variants existing in the literature. Then, we present self-stabilizing distributed protocols resilient to permanent or intermittent faults (see Section 4.2), namely fault-tolerant and self-stabilizing distributed protocols (see Section 4.2.1), Byzantine-tolerant and self-stabilizing distributed protocols (see Section 4.2.2), and strictly-stabilizing distributed protocols (see Section 4.2.3). Finally, we compare these different fault tolerance schemes in Section 4.3.

Specification In the remainder of this chapter, we assume that we have a distributed task (*a.k.a.* problem) to solve. Some classical examples of these distributed tasks are leader election (the distributed system must agreed on a distinguished vertex in a finite time), token circulation (a particular message must circulate and reach

infinitely often each vertex), mutual exclusion (all vertices must execute infinitely often a critical section with the guarantee that no two vertices execute it at the same time), and so on (see [Tel10] for some other examples).

In order to prove correctness or fault-tolerance of a given distributed protocol with respect to a task, we need a formal definition of this task. The specification of a task is a predicate that states if an execution of a distributed protocol is legitimate with respect to the task (*i.e.* if this execution is a solution of the task). Note that a given task may admit several specifications.

4.1 Tolerating Transient Fault Patterns

As mentioned previously, self-stabilization is convenient to tolerate any transient fault pattern. This section is devoted to the description of this fault tolerance scheme and its main variants (at the notable exception of those tolerating composite fault patterns that are described in Section 4.2).

Self-stabilization was defined by Dijkstra in [Dij74]. Intuitively, to be self-stabilizing, a distributed protocol must satisfy the two following properties:

Closure: there exists some configurations from which any execution of the distributed protocol satisfies the specification; and

Convergence: starting from any arbitrary configuration, any execution of the distributed protocol reaches in a finite time a configuration that satisfies the closure property.

We can use a self-stabilizing distributed protocol to perform fault-tolerance due to the following observations. At the end of a burst of transient faults, the configuration of the system may be arbitrary (recall that we do not add assumptions on the nature or the span of the transient faults, hence any vertex may be Byzantine during the fault and arbitrarily modifies its state). Then, a self-stabilizing distributed protocol ensures that after a finite time (called the convergence or stabilization time), the distributed protocol recovers from itself a correct behavior (by convergence property) and keeps this correct behavior until there is no faults (by closure property). In the same situation, a classical (*i.e.* not fault-tolerant) distributed protocol may never recover as well as a robust distributed protocol (since the fault may not be included in its tolerated range). In conclusion, a self-stabilizing distributed protocol is well-suited for tolerating any transient fault pattern.

We state below the formal definition of self-stabilization that we use in the sequel of this thesis.

Definition 4.1 (*Self-stabilization [Dij74]*)

A distributed protocol π is self-stabilizing for specification spec if and only if starting from any arbitrary configuration every execution of π contains a configuration from which every execution of π satisfies spec.

The literature related to self-stabilization is too large to do an exhaustive state-of-the-art ([Her02] lists over 500 papers but stops in referencing self-stabilizing works

in 2002). The interested reader is referred to the book of Dolev [Dol00] or the book chapter of Tixeuil [Tix09]. Note that there exists few surveys in the self-stabilizing area with the notable exceptions of a survey from Gärtner about spanning tree construction [Gär03] and a survey from Guellati and Kheddouci on independence, domination, coloring, and matching [GK10].

In the following, we quickly present main variants of self-stabilization that are not related to permanent fault tolerance (these variants are the subject of the following section). We present first some variants of self-stabilization that exhibits weaker properties (see Section 4.1.1) and then variants that exhibits stronger properties (see Section 4.1.2) than self-stabilization. Note that a survey on variants of self-stabilization is provided by Devismes, Petit, and Villain [DPV11a, DPV11b].

4.1.1 Weakening Self-Stabilization

This section presents a (non-exhaustive) survey on variants that weaken self-stabilization using different approaches. The main goal of these variants is obviously to allow to solve a larger class of problem than self-stabilization.

Pseudo-stabilization First, we present pseudo-stabilization introduced by Burns, Gouda, and Miller in [BGM93]. They remove the guarantee of reaching a configuration from which the execution satisfies the specification. Instead, a pseudo-stabilizing distributed protocol guarantees that every execution has a suffix that satisfies the specification. Formal statement of this concept follows.

Definition 4.2 (*Pseudo-stabilization [BGM93]*)

A distributed protocol π is pseudo-stabilizing for specification spec if and only if starting from any arbitrary configuration every execution of π has a suffix satisfying spec.

The main weakening with respect to self-stabilization is the following: when a portion of execution satisfies the specification, we have no guarantee that any subsequent execution satisfies the specification (due to the implicit weakening of the closure property). However, for an external observer, the execution is eventually correct that motivates pseudo-stabilization from a practical point of view when self-stabilization is not possible. For instance, [BGM93] proves that the alternating bit protocol (see Section 6.1.1) is pseudo-stabilizing.

Probabilistic stabilization Although this thesis focuses only on deterministic distributed protocols, we quickly discuss probabilistic stabilization. Probabilistic stabilizing distributed protocols are introduced in [Her90] as distributed protocols that ensure the convergence property with probability 1. Later, [DPBT04] distinguished a strong and a weak variant of probabilistic stabilization. The original definition of [Her90] corresponds to strong probabilistic stabilization while weak probabilistic stabilization weakens also the closure property since this latter is only

probabilistic. For instance, the mutual exclusion distributed protocol provided by [DHT04] falls in weak probabilistic stabilizing distributed protocols category.

Weak-stabilization Gouda introduced in [Gou01] another weakening of self-stabilization called weak-stabilization. Starting from any arbitrary configuration, a weakly stabilizing distributed protocol only ensures that there exists at least one execution that reaches in a finite time a configuration from which any execution satisfies the specification. In other words, weak-stabilization weakens the convergence property by requiring only the existence of executions that satisfies the convergence property (and not that each execution satisfies it).

The main result of [Gou01] is to establish that a weakly-stabilizing distributed protocol is in fact self-stabilizing if we consider a distributed system such that Γ is finite under a Gouda fair daemon (see Section 3.1.2). Another result related to weak-stabilization is due to [DTY08]. It proves a strong connection between weak-stabilization and probabilistic stabilization: a weakly-stabilizing distributed protocol can be automatically turned into a probabilistic stabilizing one under a probabilistic daemon (*i.e.* a distributed daemon whose choices are probabilistic).

4.1.2 Enhancing Self-Stabilization

In this section, we present some of the variants of self-stabilization that exhibit stronger fault tolerance than self-stabilization (by requiring supplementary properties). Note that we do not consider here permanent or intermittent fault tolerance issues since they are described in details in the following section of this chapter.

Snap-stabilization Bui *et al.* defined snap-stabilization in [BDPV07]. A snap-stabilizing distributed protocol ensures that, starting from any arbitrary state, any execution of the distributed protocol satisfies the specification. In other words, a snap-stabilizing distributed protocol is a self-stabilizing distributed protocol with a stabilization time of 0. The main interest of snap-stabilization is the stronger safety properties provided to the user of the distributed system. Snap-stabilization was essentially studied through the propagation of information with feedback (PIF) problem [BDPV07], the token circulation problem [PV07], or the message forwarding problem [CDV09].

Super-stabilization Super-stabilizing distributed protocols have been introduced by Dolev and Herman in [DH97] as self-stabilizing distributed protocols that moreover satisfies a safety predicate when a topological change occurs during a legitimate execution. Note that constraints on topological changes are strong since they cannot occur during the stabilization phase (in this case, the distributed protocol may never stabilize). We can generalize this approach by defining classes of self-stabilizing distributed protocols that moreover satisfies a given predicate (either during or after the stabilization phase) in spite of the occurrence of new faults (of defined nature, duration, and/or span). For instance, [JT03] proposes a shortest path spanning tree

construction that ensures moreover a property of route-preservation (intuitively, once a path is constructed towards a vertex, no message for this vertex can be lost).

Fault-containing self-stabilization With a self-stabilizing distributed protocol, even a local transient fault may cause corrections in the whole distributed system. In order to avoid such an undesirable behavior, some works [GGHP07, BDH06] define a fault-containing self-stabilizing distributed protocol as a distributed protocol that, in addition to providing self-stabilization, contains the effects of faults. This ensures that disruption during recovery from faults, is proportional to the extent of the faults. Note that this notion can be adapted to variants of self-stabilization. For instance, [DGX11] shows that some weakly-stabilizing distributed protocols of [DTY08] may be adapted to become fault-containing.

The notion of fault-containment must not be confused with containment of strict-stabilization (see Section 4.2.3) that assumes arbitrary transient and intermittent Byzantine fault patterns (recall that a fault-containing self-stabilizing distributed protocol is unable to tolerate such fault patterns).

4.2 Tolerating Composite Fault Patterns

This section aims to present existing fault tolerance schemes that are able to deal with composite fault patterns (see Section 2.4.2). Recall that a composite fault pattern gathers several classical fault patterns. In the following, we focus on tolerance to transient and permanent crash fault patterns (see Section 4.2.1) and to transient and intermittent Byzantine fault patterns (see Sections 4.2.2 and 4.2.3).

4.2.1 Fault-Tolerant Self-Stabilization

Fault-tolerant self-stabilization is the most natural way to gather robust and self-stabilizing distributed protocols properties. Indeed, this fault tolerance scheme defined simultaneously by [AH93] and by [GP93] ensures that distributed protocols satisfies closure and convergence properties of Section 4.1 in presence of any arbitrary transient and permanent crash fault pattern. In other words, a fault-tolerant self-stabilizing (FTSS for short) distributed protocol recovers a correct behavior in a finite time after transient faults in spite of crashes of a given proportion of vertices.

We can state the formal definition of fault-tolerant self-stabilization in the following way:

Definition 4.3 (*Fault-tolerant self-stabilization [AH93, GP93]*)

A distributed protocol π is f-fault-tolerant and self-stabilizing (f-ftss for short) for specification spec *if and only if starting from any arbitrary configuration every execution of π involving at most f crashed vertices contains a configuration from which every execution of π satisfies* spec.

Due to strong fault tolerance properties ensured by FTSS distributed protocols, there exists numerous impossibility results related to this fault tolerance scheme. For

example, the seminal work of [AH93] proves the impossibility of the computation of the size of the communication graph or of the leader election in an asynchronous environment (it also proposes randomized distributed protocols that fall outside the scope of this survey). On the other hand, the second work that introduces FTSS distributed protocols [GP93] proposes a transformer that turns fault-tolerant distributed protocols into FTSS ones in a synchronous distributed system. This transformer relies on a strong clock synchronization (see Section 9.1.1) FTSS distributed protocol. Concerning asynchronous distributed systems, [GP93] provides a FTSS consensus distributed protocol using a failure detector (abstraction that induces some extent of synchronism [CT96]).

Other works about FTSS distributed protocols confirm the difficulty to design deterministic FTSS distributed protocols in fully asynchronous environment. For instance [BDKM96] proves the impossibility of orienting an uniform ring in such a context. In contrast, it provides a randomized 1-ftss distributed protocol that solves this task. Finally, [BKM97b] and [BKM97a] extend some previous results by showing that it is impossible to provide a similar transformer as the one of [GP93] in an asynchronous distributed system and that it is impossible to compute the size of a large class of communication graphs using a FTSS distributed protocol. They also provide a FTSS distributed protocol for computing the ring size using failure detectors.

In a similar way that we can provide variants of self-stabilization by weakening convergence or closure property (see Section 4.1.1), we can define variants of fault-tolerant self-stabilization. In this avenue of research, Delport-Gallet *et al.* define in [DGDF10] the fault-tolerant pseudo-stabilization (FTPS for short) using the key idea of the pseudo-stabilization. Indeed, they focus on the leader election problem (that is proved impossible to solve in a FTSS way by [AH93]). Hence, they must weaken the fault tolerance model to bypass this impossibility result. The formal definition of fault-tolerant pseudo-stabilization follows.

Definition 4.4 (*Fault-tolerant pseudo-stabilization [DGDF10]*)

A distributed protocol π is f-fault-tolerant and pseudo-stabilizing (f-ftps for short) for specification spec if and only if starting from any arbitrary configuration every execution of π involving at most f crashed vertices has a suffix satisfying spec.

4.2.2 Byzantine Tolerant Self-Stabilization

Once the fault-tolerant self-stabilization was defined, it was very natural to extend the definition in order to tolerate intermittent Byzantine faults instead of permanent crash faults. We call this new fault tolerance scheme byzantine-tolerant self-stabilization (BTSS for short). To the best of our knowledge, the first work dealing with simultaneous tolerance to transient and intermittent Byzantine faults is [DW95]. It introduces a definition similar to the following.

Definition 4.5 (***Byzantine-tolerant self-stabilization [DW95]***)

A distributed protocol π is f-byzantine-tolerant and self-stabilizing (f-btss for short) for specification spec if and only if starting from any arbitrary configuration every execution of π involving at most f Byzantine vertices contains a configuration from which every execution of π satisfies spec.

The seminal work of Dolev and Welch [DW95] (also appearing in [DW04]) focuses on a variant of weak clock synchronization (see Section 9.1.1) in presence of any transient and intermittent Byzantine fault pattern. Note that they assume a complete communication graph. In this context, they provide two randomized $\frac{n}{3}$-btss distributed protocols (the first for synchronous distributed systems, the second for asynchronous ones). Unfortunately, these two distributed protocols exhibit an exponential expected convergence time.

Two other interesting results in this context are due to Daliot and Dolev in [DD05] and [DD06]. Based on a BTSS strong clock synchronization distributed protocol, [DD05] proposes a transformer that turns a Byzantine-tolerant distributed protocol into its BTSS equivalent. The obtained distributed protocol requires an eventually synchronous distributed system. The main idea of the transformer is the following: at each communication round, vertices executes a Byzantine agreement. If an inconsistency is discovered, then they execute a reset in the following round. Also in the eventually synchronous model, [DD06] provides a $\frac{n}{3}$-btss agreement distributed protocol.

Finally, note that the most studied problem in Byzantine-tolerant self-stabilizing setting was clock synchronization (and some variants), see *e.g.* [PT97, DW97, DW04, HDD06, DH07, BODH08]. Section 9.1.2 proposes a survey of these works.

The challenging combination between transient and intermittent Byzantine faults obviously leads to huge difficulties in developing BTSS distributed protocols. In particular, we are not aware of the existence of any deterministic BTSS distributed protocol for asynchronous distributed systems. In the following section, we present another fault tolerance scheme that allows the design of deterministic distributed protocols for asynchronous distributed systems resilient to any transient and intermittent Byzantine fault pattern.

4.2.3 Strict Stabilization

As emphasized by the surveys presented in the two previous sections about fault- and Byzantine-tolerant self-stabilization, the design of distributed protocols simultaneously tolerant to transient and permanent or intermittent faults is a difficult (and sometimes impossible) task, in particular in a deterministic way and in asynchronous distributed systems.

Therefore, Nesterenko and Arora introduced in [NA02] another approach to joint transient and intermittent Byzantine fault tolerance using the notion of containment. Intuitively, the idea is to provide self-stabilizing distributed protocols that moreover ensure that the effects of Byzantine faults are isolated within a region of the com-

munication graph (that is, some vertices around Byzantine ones are allowed to never reach a correct behavior while all other vertices must reach a correct behavior in a finite time). In other words, this approach is non masking both for transient faults (since there exists a period of time where all vertices may have an erratic behavior) and for Byzantine faults (since vertices around Byzantine ones may have an erratic behavior during the whole execution). In contrast, Byzantine-tolerant self-stabilization is non masking for transient fault (due to the self-stabilizing approach) but masking for Byzantine faults (since any execution must satisfies the specification after convergence in spite of occurrence of Byzantine faults, like in Byzantine-tolerance).

Nesterenko and Arora defined in [NA02] strict-stabilization in the following way. A distributed protocol is strictly-stabilizing if it is self-stabilizing and if it guarantees that there exists an (unweighted) containment radius c outside which vertices are not affected by Byzantine vertices. In other words, a strictly-stabilizing distributed protocol reaches in a finite time a configuration after which the behavior of any vertex located at a distance greater than c from any Byzantine vertex is legitimate. This approach provides tolerance to any transient and intermittent Byzantine fault pattern since the effects of transient faults are contained in time (due to the convergence property) and the effects of Byzantine faults are contained in space (due to the existence of the containment radius).

We can now provide the formal definition of strict-stabilization in the following way. For any natural number c, we define g_c as the communication subgraph of g induced by the set V_c defined by (where B denotes the set of Byzantine vertices):

$$V_c = \{v \in V | \min_{b \in B}\{dist(g, v, b)\} > c\}$$

Intuitively, V_c gathers the set of correct vertices that must eventually have a correct behavior in any execution of a strictly-stabilizing distributed protocol that has a containment radius of c.

Definition 4.6 (*Strict-stabilization [NA02]*)

A distributed protocol π is (c, f)-strictly stabilizing for specification spec if and only if starting from any arbitrary configuration every execution of π involving at most f Byzantine vertices contains a configuration from which every execution σ of π satisfies: the projection of σ on g_c satisfies spec.

The parameter c of Definition 4.6 refers to the containment radius of the strictly-stabilizing distributed protocol. The parameter f refers explicitly to the number of tolerated Byzantine vertices, while [NA02] dealt with unbounded number of Byzantine faults (that is $f \in \{0 \dots n\}$).

To prove the effectiveness of strict-stabilization, Nesterenko and Arora present in [NA02] two deterministic strictly-stabilizing distributed protocols for asynchronous distributed systems. The first one focuses on vertex coloring (that is, each vertex must decide on a color with the constraint that no two neighboring vertices decide on the same color) and exhibits a containment radius of 1. This distributed protocol

is very simple since each vertex modifies its color when it detects a conflict with one of its neighbors in order to take a color carried by none of its neighbors (the set of available colors is of size $deg(g) + 1$ such that this color choice is always possible). The containment radius of 1 can be easily deduced from the following observations. Once correct vertices execute at most one time their rule, no two correct neighbors can be in conflict (note that the distributed protocol runs under the locally central daemon) and Byzantine vertices can only trigger color changes on their neighbors (since each of them does not choose a color conflicting with the ones of its own neighbors). The second distributed protocol proposed by [NA02] focuses on dining philosophers problem (*i.e.* local mutual exclusion) and performed a containment radius of 2.

An essential question about strict-stabilization is the optimality of the containment radius for a given problem. Indeed, smaller is the containment radius of a strictly-stabilizing distributed protocol, better is its fault-tolerance. Nesterenko and Arora provides a generic impossibility result that provides a beginning of answer to this question. First, they defined the notion of r-restrictive specification. Intuitively, a specification is r-restrictive if it prevents combinations of configurations that differ by the state of two vertices that are at least r hops away. Then, they prove the following result: any strictly-stabilizing distributed protocol for a r-restrictive specification has a containment radius of at least r. Note that this result is not a characterization of the optimal containment radius. For instance, both vertex coloring and dining philosophers problems admit a 1-restrictive specification. This is sufficient to prove the optimality of the containment radius of the previously presented strictly-stabilizing distributed protocol for vertex coloring. On the other hand, a result from [CS96] proves the optimality of the containment radius of the strictly-stabilizing solution of [NA02] for dining philosophers (while this optimality cannot be deduced from the result about r-restrictive specifications).

Note that there are very few works about strict-stabilization. At our knowledge, only [SOM05] and [MT07] provide strictly-stabilizing distributed protocols before this thesis. They focus on the link coloring problem (each edge of the communication graph must receive a color with the constraint that no two adjacent edges have the same color) respectively on trees and on general communication graphs.

4.3 Summary

In this chapter, we survey numerous fault tolerance schemes in distributed systems. Classically, we distinguish robust distributed protocols (that ensure a masking fault tolerance to permanent crash or intermittent Byzantine fault patterns) and self-stabilizing distributed protocols (that ensures a non masking fault tolerance to transient fault patterns). We presented many variants (and composition) of these two approaches. We can compare these various fault tolerance schemes by the class of tolerated fault patterns and by their masking properties with respect to permanent crash or intermittent Byzantine faults and/or with respect to transient faults.

Fault tolerance scheme	Tolerated class of fault patterns	Masking for transient faults	Masking for other faults
Self-stabilization	Transient	No	–
Fault-tolerant	Permanent crash	–	Yes
Byzantine-tolerant	Intermittent Byzantine	–	Yes
FTSS / FTPS	Transient and permanent crash	No	Yes
BTSS	Transient and intermittent Byzantine	No	Yes
Strict-stabilization	Transient and intermittent Byzantine	No	No

Table 4.1: Comparison of fault tolerance schemes presented in Chapter 4

Table 4.1 presents a comparison of fault tolerance schemes presented in this chapter.

In this thesis, we concentrate on fault tolerance schemes that enhance self-stabilization with properties of permanent crash or intermittent Byzantine fault tolerance. Another way to compare this class of fault tolerance schemes is to define a relation of constraint between them. We say that a fault tolerance scheme is more constrained that another if any distributed protocol that satisfies the first satisfies the second. For instance, self-stabilization is more constrained than pseudo-stabilization since any self-stabilizing distributed protocol is also pseudo-stabilizing (by definition). Figure 4.1 sums up these relations between permanent or intermittent fault tolerance schemes in self-stabilization. Note that this relation of constraint is transitive but not total since some fault tolerance schemes are not comparable. For instance, self-stabilization and fault-tolerant pseudo-stabilization are not comparable since the first does not tolerate permanent crash faults but provides a stronger convergence property than the second. It is natural to conjecture that the more constrained is a scheme the more difficult is to provide a distributed protocol according to this scheme (this conjecture is well illustrated by impossibility results in fault-tolerant self-stabilization).

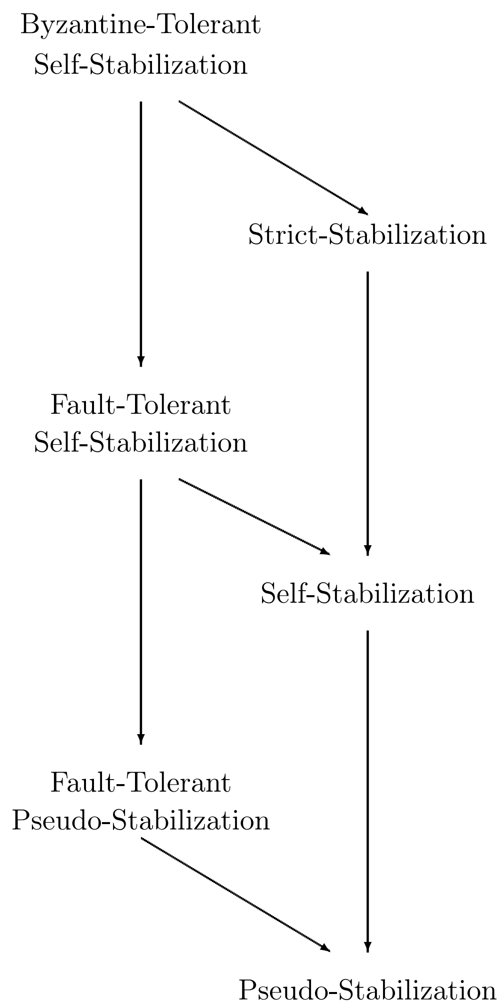

Figure 4.1: Summary of respective constraints on permanent or intermittent fault tolerance schemes in self-stabilization. An arrow from a scheme to another means that the first is more constrained than the second. Note that we remove all arrows deductible from transitivity.

Part II

Atomic Register

Chapter 5

Introduction of Part II

> The function of modeling is to arrive at descriptions which are useful.
>
> Richard Bandler and John Grinder

Contents

5.1 Problem and Related Works **72**
5.1.1 Problem 72
5.1.2 Related Works 74
5.1.3 Specification 76
5.2 Contributions of Part II **77**

In the second part of this thesis, we deal with a fundamental problem in distributed systems: the simulation of a computational model over another one. The main interest of this problem comes from the following observation: higher is the atomicity of a computational model, simpler is the design of distributed protocols but, on the other hand, lower is the realness of this distributed protocol. In order to keep the simplicity of a high atomicity computational model in a low atomicity one, an interesting solution is to design a transformer from the second to the first. In this way, one can design distributed protocols for the high atomicity computational model (enjoying its simplicity) but executes it in the low atomicity computational model (enjoying its realness) by composing it with the transformer.

For instance, [DIM97a] studies the simulation of the state model over the message passing model (see Section 2.3) in a self-stabilizing context. There exists also transformers from very specific computational models to more classical ones in order to re-use all existing distributed protocols (designed for the classical computational model) in the specific computational model. For example, [KA06] proposes a transformer from a computational model specific to wireless distributed systems (the local diffusion model with collisions) to the classical state model.

The main drawback of these model transformers is that they generally introduce an important overhead (in time and in space) with respect to a distributed protocol directly designed for the low atomicity computational model. On the other hand, they allow to prove computability equivalences between different computational model since any task solvable in the high atomicity computational model is solvable in the low atomicity one using the suitable transformer.

In this part, we focus on the simulation of a classical computational model of fault-tolerant distributed systems, the atomic register model (see *e.g.* [Lyn96]), over

the message passing model (see Section 2.3.3). The atomic register model is characterized by the fact that communications between vertices are only performed by read and write operations on shared variables (called registers) that provide atomicity [Lam86a, Lam86b] (see Section 5.1.1). This computational model transformation (also called simulation) is well-studied in distributed systems subject to permanent crash or intermittent Byzantine fault patterns (see Section 5.1.2). We propose to study this simulation in distributed systems subject to any transient and permanent crash fault pattern.

This chapter aims to define the atomic register simulation problem and to present our contribution with respect to existing works. Section 5.1 provides a survey on atomic register model simulation while Section 5.2 summarizes assumptions and contributions of Part II.

5.1 Problem and Related Works

We present in details the register model (see Section 5.1.1), provide a survey of existing simulations of atomic registers both in distributed systems subject to classical and composite fault patterns (see Section 5.1.2), and specify formally the problem solved in this part (see Section 5.1.3).

5.1.1 Problem

Registers have been introduced by Lamport [Lam86a, Lam86b] as a model of communication between vertices of a distributed system. A register is a variable (over a domain D) shared by all vertices of the distributed system that provides two operations: a read operation that returns the value of the register to the invoking vertex and a write operation that allows the invoking vertex to modify the value of the register. Using registers, it is possible to define a computational model (see Section 2.3) such that vertices only communicate by registers and in which read and write operations are assumed atomics. From now, we refer to this computational model as the register model. This computational model is extensively used in distributed system literature (see *e.g.* [Lyn96]).

Given a register, we call readers the vertices that are able to invoke the read operation of the register and writers the vertices that are able to invoke the write operation of the register. According to [Lam86a, Lam86b], we can classify registers using several criteria:

1. size of the domain of the register: a d-ary register has a domain of size d (that is $|D| = d$);
2. maximal number of writers of the register: a single-writer register has only one writer while a multi-writer register has an arbitrary number of writers;
3. maximal number of readers of the register: a single-reader register has only one reader while a multi-reader register has an arbitrary number of readers; and

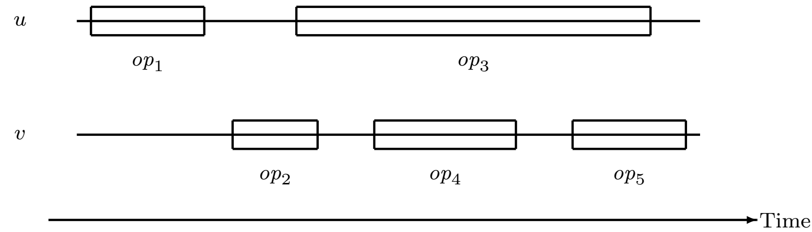

Figure 5.1: Illustartion of concurrent or consecutive operations.

4. nature of the register (defined by the value returned by a read operation).

In the following, we discuss this last criterion.

Nature of the register As readers of a register may be distinct from its writers, read and write operations may be interleaved in some executions of the distributed system. Then, we must clarify the result of read operations in such cases. Lamport [Lam86a, Lam86b] distinguishes three types of registers according to read operation properties: safe, regular and atomic. In the following, we define each of them.

In the low atomicity computational model, read and write operations on the register are not instantaneous and then need the execution of several instructions. Each operation starts when a vertex invokes it and ends when it returns. We say that an operation op_1 happens before an operation op_2 if op_1 ends before op_2 starts. Two operations op_1 and op_2 are concurrent if they satisfies: op_1 does not happen before op_2 and op_2 does not happen before op_1. Two operations op_1 and op_2 are consecutive if op_1 is the most recent operation that happens before op_2. Figure 5.1 provides an example of a set of operations performed by two vertices u and v. We can observe that op_1 happens before op_2 while op_3 is concurrent with op_2, op_4, and op_5. Operations op_2 and op_4 are consecutive.

Now, we can define properties of registers in the following way:

Safe register: A safe register can be written by one writer only. Moreover, a read operation on such a register returns the current value of the register if no write operation is concurrent with that read. In case of concurrency with a write operation, the read operation can return any value of D, the domain of the register (note that a read operation concurrent with a write operation can return a value that has never been written).

Regular register: A regular register can have any number of writers. A regular register is a safe register (a read operation not concurrent with a write operation returns the actual value of the register) such that a read operation concurrent with a write operation returns either the old value (the value of the register before the write operation starts) or the new value (the value written by the write operation) of the register. Hence, a regular register is stronger than a safe register since the value returned in presence of concurrent write operations is no longer arbitrary.

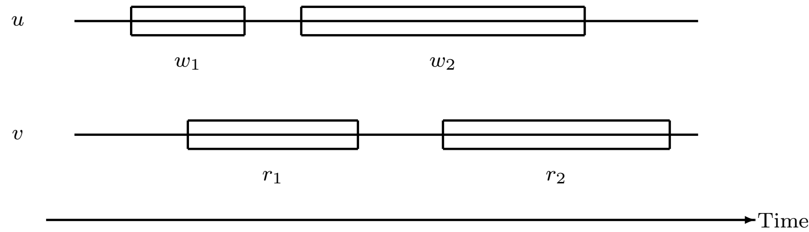

Figure 5.2: If r_2 returns the value written by w_1 and r_1 returns the value written by w_2, we have a new/old inversion.

Nevertheless, a regular register can exhibit new/old inversions. Consider two consecutive read operations r_1, r_2 and two consecutive write operations w_1, w_2 such that r_1 is concurrent with both w_1 and w_2 and r_2 is concurrent only with w_2 (see Figure 5.2). The regularity property allows r_2 to return the value written by w_1 and r_1 to return the value written by w_2.

Atomic register: An atomic register is a regular register without new/old inversion. For instance, for the execution illustrated by Figure 5.2, the atomicity of the register implies that, if r_1 returns the value written by w_2, then r_2 must return the same value. This means that an atomic register is such that all its read and write operations appear as if they have been executed sequentially, this sequential total order respecting the real time order of the operations.

Classification of registers As highlighted by [Lam86a, Lam86b], this set of definitions allows us to define a partial order over registers. We say that a register is stronger than another one if it provides stronger properties. For example, a multi-writer multi-reader binary regular register is stronger than a multi-writer multi-reader binary safe register. Note that some registers are not comparable using this order.

5.1.2 Related Works

As register computational model is largely used in distributed systems (see [Lyn96] for some examples), the simulation of registers over lower atomicity computational model was well-studied. Due to its stronger properties, the multi-writer multi-reader atomic register is a kind of "holy grail" of this avenue of research.

Due to the number of related works, it is quite impossible to do an exhaustive survey of register simulation. In the following, we focus on atomic register simulation (over message passing model) in distributed systems subject to various fault patterns. To construct an atomic register, there exists mainly two methods:

- simulate an atomic register over a weaker register computational model (safe, regular, single writer,...); or
- directly simulate the atomic register over message passing model.

However, note that these two approaches are not exclusive since we can simulate a given register computational model over message passing model and then use it to simulate another stronger register computational model. We present works related to these two approaches under various assumptions of fault-tolerance in the following.

Simulation over a weaker register computational model A well-studied approach to provide atomic registers is to construct it on the top of a weaker register (according to the partial order defined in Section 5.1.1). In the following, we describe main results on this topic in distributed systems subject to various fault patterns (note that there also exists such simulations in fault-free distributed systems, see *e.g.* [Lam86b]).

First, atomic register simulation was well-studied in a fault-tolerant context. We can cite a few examples of simulation from a weaker register computational model. There exists simulations that focuses on the nature of the register as [HV95] that simulates a single-writer multi-reader atomic register from a single-writer multi-reader regular register. Some other transformations look at the number of readers and of writers as [Blo88] that constructs a 2-writer multi-reader atomic register from two single-writer multi-reader atomic registers. Another possible simulation is about the domain of the register as [CKW00] that simulates a d-ary regular register using $\frac{d(d-1)}{2}$ binary regular registers.

Regarding tolerance to transient fault patterns, only few works deal with self-stabilizing simulation of register. Whereas they fall outside the scope of this survey on atomic registers, we can cite [DH01] that provides self-stabilizing simulations of safe and regular registers and [JH09] that emulates a self-stabilizing regular register from a safe one. At our knowledge, the only self-stabilizing simulation of an atomic register from a weaker register is due to [IS92] that provides space-optimal simulations of a multi-writer multi-reader register respectively from a single-writer single-reader atomic register and from a single-writer multi-reader atomic register.

Finally, [HPT02] is interested in atomic register simulation in distributed systems subject to transient and permanent crash fault patterns. First, it proves the impossibility to provide a FTSS single-writer single-reader regular or atomic register simulation over a single-reader single-writer safe register computational model. Note that this result does not hold for fault-tolerant (and not self-stabilizing) simulations. Then, it designs a FTSS single-writer single-reader regular or atomic register simulation over a 2-reader single-writer safe register computational model.

Simulation over message passing model The second main way to design an atomic register simulation is to directly write this simulation in a lower atomicity computational model (*e.g.* the message passing model). As previously, we interest in simulations tolerating various fault patterns.

Existing works in this area mainly deal with tolerance to permanent crash fault patterns. The first simulation of a single-writer multi-reader atomic register in such

an environment is due to Attiya, Bar-Noy, and Dolev in [ABND90, ABND95] and was nicknamed the ABD simulation. This simulation assumes that a majority of vertices remains correct in any execution (that is, $n > 2f$ where f is the maximal number of crashed vertices) and introduces a general tool that was extensively re-used: the communication by quorum. Essentially, the key ideas of the simulation follow. Each vertex stores a copy of the register (the actual value and a version number). When the writer invokes its write operation, it sends the new value (with the new version number) to all other vertices. The write operation ends when the writer receives $n-f$ acknowledgments (each vertex that acknowledged the new value updates its own copy of the register). When a reader invokes its read operation, it asks to all vertices their current value of the register. When the reader receives $n-f$ answers, at least one of them contains the last value written by the writer and has the greatest version number. Hence, the reader can return this value. To avoid new/old inversions, the reader must write the value just before return it. This simulation is first proposed with unbounded version numbers (natural numbers with the classical total order) and then adapted to deal with bounded time-stamps (see *e.g.* [IL93] and Section 6.2) in order to use only a finite memory.

This simulation was further declined to deal with more challenging environments. For instance, [GLS10] adapts it to dynamic distributed systems and [MR98] generalizes the quorum communication to distributed systems subject to intermittent Byzantine fault pattern. For more details on these variations of the ABD simulation, the interested reader is refereed to [Att10].

If we focus now on register simulation over message passing model tolerating other fault patterns, Dolev, Israeli, and Moran present in [DIM97a] a self-stabilizing simulation of a single-writer, single-reader atomic register. In particular, note that we are not aware of the existence of any atomic register simulation tolerant to transient and permanent crash fault patterns.

5.1.3 Specification

We can now formally specify the problem that we solve in this part. We choose to focus on single-writer multi-reader atomic register simulation over message passing model. Consequently, we assume that there exists a distinguished vertex, the writer that is supplied with two operations: *read* and *write* while other vertices, the readers, are supplied with only one operation: *read*. Each *read* invocation needs no parameter and returns a value from D, the domain of the register. Each *write* invocation needs a parameter from D and returns no value. We say that a value v is written to the register when the operation *write*(v) returns.

Specification 5.1 (1-*writer n-reader atomic register simulation* $spec_{ARS}$)

An execution σ satisfies $spec_{ARS}$ if and only if it complies with the following two properties:

(Regularity) *Each read operation returns either the value written by the most recent write operation that happens before it or a value written by a con-*

current write operation.

(No new/old inversion) *If a read operation r returns a value written by a concurrent write operation w then no read operation that happens after r returns a value written by a write operation that happens before w.*

5.2 Contributions of Part II

Position of Part II As pointed out in Section 5.1.2, we are not aware of the existence of any work studying the simulation of an atomic register over a lower atomicity computational model in distributed systems subject to transient and permanent crash fault patterns. The main contribution of the Part II of this thesis is to remedy to this fact by providing such a simulation.

More precisely, we assume from now that our distributed system is modeled by a complete and identified communication graph and that it may be subject to any (k, f, ℓ)-transient and permanent crash fault pattern with $n > 2f$. We choose to provide a single-writer multi-reader atomic register simulation over the message passing model presented in Section 2.3.3.

Overview of Part II The contribution of the Part II of this thesis is twofold. First, we need to design two tools that are necessary to our simulation. However, we believe that these tools are sufficiently independent to be useful in another context. Hence, Chapter 6 presents the following results:

1. a self-stabilizing communication protocol over unreliable and non-FIFO communication links that performs the optimal fault resilience; and
2. a stabilizing bounded labeling system that improves existing bounded labeling systems (like the one of [IL93]) by tolerance to arbitrary initial configuration.

Second, using the two preliminary tools of Chapter 6, we show that the ABD simulation needs only few changes to handle transient and permanent crash fault patterns instead of permanent crash fault patterns. Indeed, Chapter 7 proves that the bounded memory version of the ABD simulation can be turned into a fault-tolerant pseudo-stabilizing single-writer multi-reader atomic register simulation.

Results of Chapter 6 appear in Information Processing Letters [DDPBT11b], in proceedings of the 24th International Symposium on Distributed Computing (DISC 2010) [AAD+10], and in proceedings of the 13èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (Algotel 2011) [DDPBT11a]. Regarding Chapter 7, preliminary versions of the presented simulation are published in proceedings of the 24th International Symposium on Distributed Computing (DISC 2010) [AAD+10] and of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011) [AAD+11].

Chapter 6

Preliminaries

> Be convinced that, if man were able to reach the end without preparatory studies, such studies would not be preparatory but tiresome and utterly superfluous.
>
> Maimonides

Contents

6.1 Data-Link Protocol **79**
- 6.1.1 Problem and Related Works 80
- 6.1.2 Specification 81
- 6.1.3 Lower Bounds 83
- 6.1.4 Optimal Solution 85
- 6.1.5 Correctness Proof 88

6.2 Bounded Labelling Systems **93**
- 6.2.1 Problem and Related Works 93
- 6.2.2 Solution 96

In this chapter, we study two different tools that are necessary to our solution of atomic register simulation of Chapter 7. These two tools are independent of our problem and may be re-used in another context.

The first tool we present in Section 6.1 is a communication primitive. Its goal is to simulate better properties than those exhibited by communication channels of our message passing model (see Section 2.3.3). In particular, our communication primitive minimizes the number of undesired lost, duplication, creation, and re-ordering of messages.

The second tool presented in Section 6.2 is a new bounded labeling system that provided properties suitable for a self-stabilizing setting whereas any existing bounded labeling scheme cannot handle an arbitrary initial configuration.

6.1 Data-Link Protocol

Recall that the message passing communication model presented in Section 2.3.3 assumes that communication channels are asynchronous, not reliable (but fair), bounded, and not-FIFO. Moreover, the arbitrary initial configuration due to transient faults may lead communication channels to contain fake messages.

The goal of our communication primitive, that we called a data-link protocol, is to provide the best possible properties on message delivery in this challenging environment.

6.1.1 Problem and Related Works

As self-stabilization is usually considered a hard property to satisfy, most related works used the state model (see Section 2.3.2) in which any vertex can determine the current state of every neighbors (and update its own state accordingly) in an atomic manner. Asynchronous message passing is a more realistic way, compared to the state model, for the communication of vertices in distributed systems. In such settings vertices communicate by exchanging messages, where sending and receiving message are two separate atomic actions (see Section 2.3.3). Transformers for distributed protocols from state model to message passing model, assuming the existence of FIFO communication channels, have been suggested, see *e.g.* [DIM93, Dol00]. At the core of those transformers are the *data-link* protocols, that permit to reliably exchange information between neighboring vertices in the message passing model. In addition, several self-stabilizing distributed protocols (*e.g.* [DT06, AAD+10]) that are directly written in the message-passing model use an underlying data-link protocol as a building block.

Related Works The most studied data-link protocol, namely the alternating bit protocol (ABP), was proved to satisfy some stabilization properties [AB93, DIM97b, BGM93]: in any execution of ABP, there exists a suffix that satisfies the specification (*i.e.* the ABP is pseudo-stabilizing [BGM93], see Section 4.1). However, the impossibility to bound the amount of time before this suffix is reached makes the ABP unsuitable for most tasks. In [GM91, DIM93], Gouda and Multari and Dolev, Israeli, and Moran independently prove that for a wide class of problems (including data-link construction) guaranteeing self-stabilization when communication channels have unbounded initial capacity requires some kind of unboundedness in the protocol (either unbounded memory in [GM91], the existence of some aperiodic function [AB93], or access to a probabilistic variable [AB93]). In other words, those approaches require to implement unbounded capacities with finite memory, and are thus unlikely to be actually used in real distributed systems. Also, the expected time before reaching a stable global state depends on the initial contents of communication channels, and is thus unbounded.

Most recent works took the more realistic approach of assuming communication channels with bounded initial capacity. The token passing protocol in [DIM97b] can be used as a self-stabilizing ABP on bounded communication channels and only uses bounded memory. Howell *et al.* [HNM99] provide another data-link protocol over bounded communication channels with the additional assumption that the underlying communication channels are unreliable (*i.e.* they may loose or duplicate messages). Later, Varghese [Var00] presented self-stabilizing solutions for a wide class of problems (including data-link) in the same setting using only bounded memory.

The FIFO ordering is crucial for the stabilization since solution relies on the fact that a sequence number that is unique in the system is eventually generated and flushes every stale message in transit. A common drawback of all aforementioned self-stabilizing data-link solutions is that they assume a FIFO order on messages in the underlying communication channels.

A notable exception are the protocols provided in [BKM97a] that assumed a non-FIFO message passing model. The main difference with our approach stands in the fact that their distributed system is enhanced with some failure detector whereas we assume a fully asynchronous distributed system.

Another drawback of previously mentioned self-stabilizing data-link solutions is that they do not consider the quantitative impact of faults from the perspective of the upper layer distributed protocol (*i.e.* the layer that actually uses the data-link). Indeed, starting from an arbitrary global state where communication channels may initially contain messages of arbitrary content, being able to bound the number of sent messages that are lost or duplicated, or the number of fake messages that are actually delivered to the destination is a very important matter. The bound on the number of faulty messages delivered by a data-link protocol is an important criteria for the data-link usability in larger application, in order to ensure the fault-resiliency of the global protocol stack. To our knowledge, only [DT06, DDNT10] addresses, to some extent, this concern. A snap-stabilizing data-link (and global reset) for bounded capacity FIFO communication channels appears in [DT06]. In [DDNT10] a snap-stabilizing solution to the propagation of information with feedback (PIF) problem is presented. The solution can be seen as a data-link protocol when reduced to a 2-vertex system. Snap-stabilization implies that any message that is actually sent by the sender vertex is eventually received by the receiver vertex, so the number of lost messages is 0. However, we cannot provide bounds on the number of duplications of a given message or on the number of ghost messages (that is, messages that are not sent by the sender but received by the receiver due to the arbitrary content of communication channels in the initial configuration). Concerning the self-stabilizing protocols, only an order of magnitude on those numbers can be inferred from the stabilization time (if m messages sent or received are required to enter a legitimate global state from any arbitrary initialization, then at most m messages could be lost, duplicated, or wrongly delivered). To our knowledge, the question of fault-resilience optimality (with respect to lost, duplicated, ghost or reordered messages) for data-link protocols has never been raised before, although it has important practical consequences.

6.1.2 Specification

The specification we provide in this section is borrowed from [Lyn96] but we adapt it to the self-stabilizing context. In particular, we introduce the idea to bound the number of lost, duplicated, ghost and re-ordered messages by some constants.

Consider a system of two vertices v_i and v_j. A distributed application needs to send some messages from v_i to v_j. We say that the application layer of v_i sends

a message when it requests the communication protocol to carry this message to v_j. This message is delivered to v_j when the communication protocol releases this message to the application layer of v_j. A ghost message is a message delivered to v_j whereas v_i did not send it previously (due to the arbitrary content of communication channels in the initial configuration). A duplicated message is a message that is delivered several times to v_j whereas v_i sent it only once. A message is lost when v_i sends it but v_j never delivers it. A message m is reordered when it is delivered to v_j before a message m' whereas m has been sent after m' by v_i. Intuitively, the goal of a data-link protocol is to provide a communication black box that ensures there is no lost, duplicated, ghost, or reordered messages during any execution. In the sequel, we formally specify the data-link problem.

We associate to any execution σ the sequence $S(\sigma) = m_0 m_1 m_2 \ldots$ of messages sent by v_i in σ and the sequence $R(\sigma) = m'_0 m'_1 m'_2 \ldots$ of messages delivered to v_j in σ. Note that we consider that all sent messages are different (even if their actual content are identical, we can distinguish them as external observer of the distributed system). We introduce the following notations. For any sequence W and any integers i and j, W^j is the prefix of W of length j and W_i is the suffix of W such that $W = W^{i-1}.W_i$ (where . denotes the concatenation operator). The notation ε denotes the empty sequence. For example, $R(\sigma)^0 = \varepsilon$ for any execution σ. For any message m, we define the sequence m^* as the repetition of m an arbitrary number of times (possibly 0). For any sequence W, the sequence W^* is the result of the application of the * operator to each message of W. For instance, if $W = m_1 m_2 m_3$, then $W^* = m_1^* m_2^* m_3^*$.

We can now state the specification of the data-link problem borrowed form [Lyn96] using our notations.

Specification 6.1 (*Data-link communication spec$_{DLC}$*)

An execution σ satisfies $spec_{DLC}$ over c-bounded channels (with v_i and v_j being respectively the sender and the receiver) if:

1. *No message sent by v_i is lost.*
2. *No message delivered to v_j is a duplicated message.*
3. *No message delivered to v_j is a ghost message.*
4. *No message delivered to v_j is a reordered message.*

Note that a self-stabilizing distributed protocol for $spec_{DLC}$ ensures that only a finite number of messages are lost, duplicated, ghost, or reordered starting from any configuration but does not ensure any bounds on these numbers. From the application layer viewpoint, these guarantees are not sufficient. Therefore, we introduce the notion of message performance of a self-stabilizing distributed protocol for $spec_{DLC}$. Intuitively, a self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels has a $(\alpha, \beta, \gamma, \delta)$-message performance if the number of lost, duplicated, ghost, and re-ordered messages in any execution are respectively bound by α, β, γ, and δ. The formal definition follows.

Definition 6.1 (***Message performance for spec$_{DLC}$***)

For any non negative integers α, β, γ, and δ, a self-stabilizing distributed protocol π for spec$_{DLC}$ over c-bounded channels (with v_i and v_j being respectively the sender and the receiver) has a $(\alpha, \beta, \gamma, \delta)$-message performance if for any execution σ of π satisfies the following properties starting from an arbitrary configuration:

- ***α-Loss:*** *The first α messages sent by v_i (in the worst case) may be lost.*

$$\exists a \leq \alpha, \forall m \in S(\sigma)_a, m \in R(\sigma)$$

- ***β-Duplication:*** *The first β messages delivered to v_j (in the worst case) may be duplicated ones.*

$$\exists b \leq \beta, \forall m \in S(\sigma), \left|\{m'_i = m | m'_i \in R(\sigma)\}\right| > 1 \Rightarrow m \in R(\sigma)^b$$

- ***γ-Creation:*** *The first γ messages delivered to v_j (in the worst case) may be ghost messages.*

$$\exists c \leq \gamma, \forall m \in R(\sigma), m \notin S(\sigma) \Rightarrow m \in R(\sigma)^c$$

- ***δ-Reordering:*** *The first δ messages delivered to v_j (in the worst case) may be reordered.*

$$\exists d \leq \delta, R(\sigma)_d \in S(\sigma)^*$$

In the following section, we show that it is impossible to provide a self-stabilizing distributed protocol for *spec$_{DLC}$* that has a $(\alpha, \beta, \gamma, \delta)$-message performance with $\beta = 0$, $\gamma = 0$, or $\delta = 0$. Then, we can deduce that a self-stabilizing distributed protocol for *spec$_{DLC}$* that has a $(0, 1, 1, 1)$-message performance achieves optimal fault-resiliency. The above definitions imply that such a communication protocol ensures that either $R(\sigma) = S(\sigma)$ or $R(\sigma) = m.S(\sigma)$ (where m is an arbitrary message, it may be present in $S(\sigma)$ and . denotes the concatenation operator) for any execution σ. In other words, the sequence of received messages by v_j is identical to the sequence of emitted messages by v_i excepted the first delivery in the worst case.

6.1.3 Lower Bounds

In this section, we propose three impossibility results related to the possible values for the parameters β, γ, and δ. We prove that the lower bounds for β, γ, and δ parameters is 1. These results confirm the claim that the protocol we propose is optimal since it implements a self-stabilizing distributed protocol for *spec$_{DLC}$* with a $(0, 1, 1, 1)$-message performance.

Theorem 6.1

There exists no self-stabilizing distributed protocol for spec$_{DLC}$ over c-bounded

channels with a $(\alpha, \beta, \gamma, \delta)$-message performance such that $\gamma = 0$.

Proof: By contradiction, let π be any self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels with a $(\alpha, \beta, 0, \delta)$-message performance. By definition π must have an instruction that delivers messages to the receiver vertex. As the program counter may be corrupted and channels may contain up to c ghost messages in the initial configuration, the receiver vertex may execute this instruction during the first action of an execution σ. In consequence, the first message of $R(\sigma)$ may be a ghost message m. Hence, we can assume that $R(\sigma)^1 = m$.

It is possible to construct the execution σ such that $m \notin S(\sigma)$. In conclusion, we have: $\exists m \in R(\sigma), m \notin S(\sigma) \wedge m \notin R(\sigma)^0 = \varepsilon$ (recall that ε denotes the empty sequence). This is contradictory with the 0-Creation property of π and implies that $\gamma \geq 1$ for any self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels. ■

Theorem 6.2

There exists no self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels with a $(\alpha, \beta, \gamma, \delta)$-message performance such that $\beta = 0$.

Proof: By contradiction, let π be any self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels with a $(\alpha, 0, \gamma, \delta)$-message performance. Following Theorem 6.1, we have $\gamma > 0$. This implies that the first message delivered to v_j in an execution σ by π may be a ghost message m. Hence, we can assume that $R(\sigma)^1 = m$.

It is possible to construct an execution σ such that the first (real) message sent by v_i to v_j and delivered to v_j by π is the same message m. This message has been sent by v_i only once but has been delivered to v_j at least twice. In conclusion, we have: $\exists m \in S(\sigma), \left|\{m'_i = m | m'_i \in R(\sigma)\}\right| > 1 \wedge m \notin R(\sigma)^0 = \varepsilon$ (recall that ε denotes the empty sequence). This is contradictory with the 0-Duplication property of π and implies that $\beta \geq 1$ for any self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels. ■

Theorem 6.3

There exists no self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels with a $(\alpha, \beta, \gamma, \delta)$-message performance such that $\delta = 0$.

Proof: By contradiction, let π be any self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels with a $(\alpha, \beta, \gamma, 0)$-message performance. Following Theorem 6.1, we have $\gamma > 0$. This implies that the first message delivered to v_j by π in an execution σ may be a ghost message m. Hence, we can assume that $R(\sigma)^1 = m$.

It is possible to construct the execution σ such that $S(\sigma)^{\alpha+2} = m_0 m_1 \dots m_{\alpha-1} m_\alpha m$ and $\forall i \in \{0, \dots, \alpha\}, m_i \neq m$. As π satisfies the α-Loss and the 0-Reordering properties, it follows that $\exists i \in \{0, \dots, \alpha\}, R(\sigma)^1 = m_i$ (otherwise, either π lost at least $\alpha + 1$ messages or reordered at least one message, that is contradictory). As $m_i \neq m$, we obtain a contradiction that shows that $\delta \geq 1$ for any self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels. ■

In the following, we present a distributed protocol that is optimal with respect to message performance parameters. That is, our distributed protocol is self-stabilizing for $spec_{DLC}$ over c-bounded channels with a $(0, 1, 1, 1)$-message performance.

6.1.4 Optimal Solution

This section aims to present in details our self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels with a $(0, 1, 1, 1)$-message performance.

Key ideas of the distributed protocol The rationale of the distributed protocol consists in adding safety extensions to the well-known alternating bit protocol (*a.k.a.* ABP). The concept used in the design of the data-link protocol is to let the sender use a mechanism based on the capacity c of communication channels so that the sender can ensure the execution of an operation in the receiver side. More precisely, the receiver acts only upon receiving a packet from the sender. The sender may repeatedly send a particular packet, and each time the receiver receives a packet it acknowledges the packet arrival.

First, the receiver can deliver a message only if $c+1$ copies of this message have been previously received: this ensures that at least one of them is genuine (*i.e.* was actually sent by the sender). Moreover, a message is delivered only if the expected bit alternates with the one of the previously received message (similarly to the ABP) in order to ensure that no message is duplicated. Indeed, the sender may still send copies of the message with the same alternating bit value until it receives a sufficient number of acknowledgments.

Second, the sender will expect for each message sent at least $3c+2$ acknowledgments with a matching alternating bit. As up to c acknowledgments could be ghost, this implies that $2c+2$ of these acknowledgments were actually sent by the receiver. One such acknowledgment could be sent by the receiver due to bad initialization, c of them could be due to c initial ghost messages in the reverse direction, and the remaining $c + 1$ can only originate from genuine messages from the sender, that triggered a delivery at the receiver.

At this stage, the distributed protocol does not ensure the 0-Loss property due to the use of the alternating bit. Indeed, if the alternating bit values of the sender and of the receiver are not synchronized at the first delivery, the receiver drops the first message. To avoid this message loss, the sender alternates between actual messages and synchronization messages. In other words, to send a message m, the sender first sends a synchronization message (denoted by $< SYNCHRO >$) until it receives $3c + 2$ acknowledgments of this synchronization message and then send the actual message m until it receives $3c+2$ acknowledgments of m. It follows that only the synchronization message may be lost and the actual message is always delivered to the receiver.

General organization of the system Our system is organized as follows. The application layer generates messages to be send from v_i to v_j. To perform this

goal, it invokes our data-link distributed protocol. Furthermore, this layer invokes functions that use communication primitives provided by the physical channel (send and receive, see Section 2.3.3).

The architecture of our system is summarized in Figure 6.1. In more details, the data-link distributed protocol is composed of two functions: $\mathcal{SDL}$**-Send** (which is executed on the sender side) and $\mathcal{SDL}$**-Receive** (which is executed on the receiver side). When the application layer on the sender side wants to send a message m, it invokes $\mathcal{SDL}$**-Send(m)**. Note that $\mathcal{SDL}$**-Send** procedure is blocking, that is if $\mathcal{SDL}$**-Send** is already in execution, the application layer waits its termination whereas the $\mathcal{SDL}$**-Receive** function is always executed on the receiver side. When the $\mathcal{SDL}$**-Receive** function has a message to deliver at the application layer on the receiver side, it executes **DeliverMessage(**m**)** that transmits m to the application layer. When the $\mathcal{SDL}$**-Receive** function wants to discard a synchronization message (since this kind of messages is useless to the application layer), it uses the **DropMessage** function that only deletes the message. Finally, each delivered message is acknowledged to the application layer on the sender side by **DeliverAck(**m**)**.

Functions $\mathcal{SDL}$**-Send** and $\mathcal{SDL}$**-Receive** must interact with the physical channel in order to exchange messages. For this, we design some functions that use communication primitives of the communication channel (send and receive).

First, we provide two operations to send a message or an acknowledgment, respectively **SendMessage**($\{m, ab\}$) and **SendAck**($\{m, ab\}$) where m is a message and ab its alternating bit value. These operations put $\{m, ab\}$ (respectively its acknowledgment) in the channel (recall that if this operation leads to more than c messages in the channel, one of them is arbitrarily deleted, see Section 2.3.3). These two send functions are described in Protocol 6.1.

Second, we provide two operations to receive a message or an acknowledgment, respectively **ReceiveMessage**() and **ReceiveAck**($\{m, ab\}$) where m is the message and ab its alternating bit value. On the receiver side, **ReceiveMessage**() is executed when the vertex wants to obtain a message from the communication channels. It returns the first message delivered by the communication channel. On the sender side, **ReceiveAck**($\{m, ab\}$) is executed by the data-link protocol and does polling. That is, it checks whether the first waiting acknowledgment in the channel (if any) matches with an acknowledgment of the parameter $\{m, ab\}$. It returns *true* if this is the case, *false* otherwise. In any case, the first waiting acknowledgment (if any) is deleted from the channel. These two receive functions are described in Protocol 6.2.

Detailed presentation of the protocol Our self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels (with a $(0, 1, 1, 1)$-message performance) $\mathcal{SDL}$ is presented in Protocol 6.3. In the following, we provide details about the two functions $\mathcal{SDL}$**-Send** and $\mathcal{SDL}$**-Receive**.

The function $\mathcal{SDL}$**-Send** takes a message m as parameter and stores the current alternating bit value in the variable ab. First, it alternates the value of ab (line

Protocol 6.1 Send functions used by our data-link protocol.

SendMessage	**SendAck**
input: $\{m, ab\}$: message **output:** No output 01: send($\{m, ab\}$,$\overrightarrow{v_i v_j}$)	**input:** $\{m, ab\}$: message **output:** No output 01: send($\{m, ab\}$,$\overleftarrow{v_i v_j}$)

Protocol 6.2 Receive functions used by our data-link protocol.

ReceiveMessage	**ReceiveAck**
input: No input **output:** The first message delivered by the communication link to the invoking vertex **persistent variable:** *last_message*: variable that can store any message or the null message $\perp$ 01: $last_message := \perp$ 02: while $last_message = \perp$ do 03: receive($last_message$,$\overrightarrow{v_i v_j}$) 04: return $last_message$	**input:** $\{m, ab\}$: message **output:** A boolean that indicates if the first acknowledgment delivered to the invoking vertex maps to the input $\{m, ab\}$ **persistent variable:** *last_ack*: variable that can store any acknowledgment or the null acknowledgment $\perp$ 01: $last_ack := \perp$ 02: while $last_ack = \perp$ do 03: receive($last_ack$,$\overleftarrow{v_i v_j}$) 04: if $last_ack = \{m, ab\}$ then 05: return *true* 06: else 07: return *false*

01) before sending a synchronization message (line 02) using an auxiliary function **AuxiliarySend**. Then, lines 03 and 04 repeat these instructions with the message m. Once the last invocation of **AuxiliarySend** returns, it delivers to the application layer the acknowledgment of m using **DeliverAck**. Now, let us describe the auxiliary function **AuxiliarySend**. This function repeatedly (while loop of line 02) sends its parameter message m (line 03) until receiving $3c + 2$ acknowledgment for this message (line 04-05).

The function $\mathcal{SDL}$**-Receive** takes no parameter and uses two variables. The first one is the alternating bit value of the last delivered or dropped message stored in *last_delivered* and the second one is a queue Q that stores the number of receptions of at most $c + 1$ different messages. Each element of this queue is a 3-tuple $(m, ab, count)$, where m is a message, ab is an alternating bit value, and $count$ is an integer denoting the number of packets (m, ab) received for the corresponding m and ab since the last **DeliverMessage** or **DropMessage** occurred. The queue [] operator takes a message m and a boolean b as operands, and either enqueues $(m, ab, 0)$ (if $(m, ab, *)$ is not present in Q, then if the queue contained $c+1$ elements, the last element of the queue is dequeued) or returns a pointer to the $count$ value associated to m and ab in Q. Any time a tuple value is changed in the queue, this tuple is

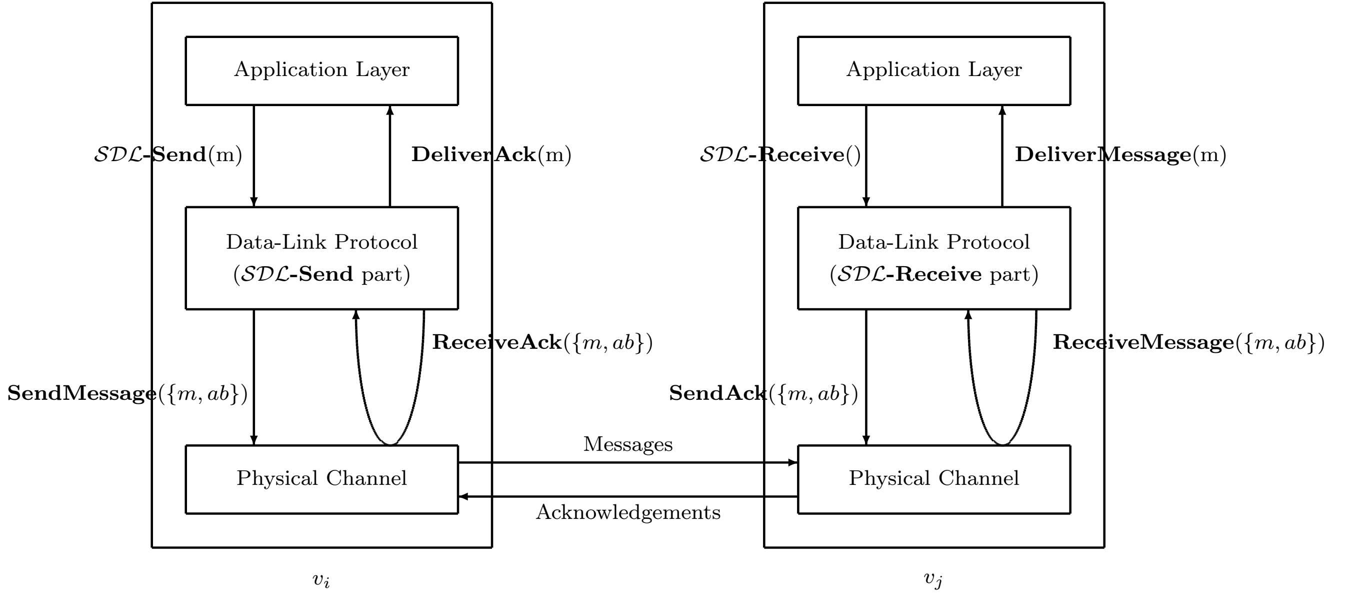

Figure 6.1: General organization of our data-link distributed protocol.

promoted at the top of the queue (in order to keep in memory the $c+1$ latest received messages), and the size of the queue does not change. The $\perp$ assignment to a queue Q denotes the fact that Q is emptied. At each reception of a message (m, ab) (line 02), the corresponding entry in the queue is updated (or created if needed) by line 03. If v_j already received $c+1$ copies of m since the last **DeliverMessage** or **DropMessage** occurred (test on line 04) then the queue is emptied (line 11). Moreover, if the alternating bit value of the message is different from *last_delivered* (test on line 05), then the message is either delivered with **DeliverMessage** (line 07) or dropped with **DropMessage** (line 09) depending if it is a synchronization message or not (test on line 06). Then, the *last_delivered* value is updated by line 10. Finally, in any case, the message is acknowledged to the sender with line 12.

6.1.5 Correctness Proof

In this section, we prove the correctness of $\mathcal{SDL}$. More precisely, we prove that $\mathcal{SDL}$ satisfies the 0-Loss property (see Lemma 6.3), the 1-Duplication property (see Lemma 6.4), the 1-Creation property (see Lemma 6.5), and the 1-Reordering property (see Lemma 6.6) starting from any configuration. By definition, this implies that $\mathcal{SDL}$ is a self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels with a $(0, 1, 1, 1)$-message performance.

The key ideas of this proof follow. First we prove that, in any execution, at most the first message parameter of **AuxilliarySend** is lost by $\mathcal{SDL}$ (see Lemmas 6.1 and 6.2). This loss happens only when alternating bits of the receiver and of the sender are not synchronized. Nevertheless, the management of the queue on the receiver side and the counting of messages and acknowledgements ensure us that these alternating bits are synchronized after the end of the first invocation

Protocol 6.3 $\mathcal{SDL}$, a self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels with a $(0, 1, 1, 1)$-message performance

$\mathcal{SDL}$-Send

input:
m: message to be sent
persistent variable:
ab: boolean that states the current alternating bit value

01: $ab := \neg ab$
02: **AuxiliarySend** $(< SYNCHRO >, ab)$
03: $ab := \neg ab$
04: **AuxiliarySend** (m, ab)
05: **DeliverAck** (m)

AuxiliarySend

input:
m': message to be sent
ab: boolean that states the alternating bit value associated to m'
variable:
ack: integer denoting the number of acknowledgments received for the current ab value

01: $ack := 0$
02: while $ack < 3c + 2$ do
03: **SendMessage**$(\{m', ab\})$
04: if **ReceiveAck**$(\{m', ab\})$ then
05: $ack := ack + 1;$

$\mathcal{SDL}$-Receive

persistent variables:
$last_delivered$: boolean that states the alternating bit value of the last delivered message
Q: queue of size $c + 1$ of 3-tuples $(m, ab, count)$, where m is a message, ab is an alternating bit value, and $count$ is an integer denoting the number of packets (m, ab) received for the corresponding m and ab since the last **DeliverMessage** or **DropMessage** occurred.

01: while $true$ do
02: $\{m, ab\}$:= **ReceiveMessage**()
03: $Q[m, ab] := min(Q[m, ab] + 1, c + 1)$
04: if $Q[m, ab] \geq c + 1$ then
05: if $last_delivered \neq ab$ then
06: if $m \neq < SYNCHRO >$ then
07: **DeliverMessage**(m)
08: else
09: **DropMessage**(m)
10: $last_delivered := ab$
11: $Q := \perp$
12: **SendAck**$(\{m, ab\})$

of **AuxilliarySend**. Then, the 0-Loss property (Lemma 6.3) follows since any actual message is always preceded by a $< SYNCHRO >$ message. We prove the 1-Duplication property (Lemma 6.4) by the use of the alternating bit. The 1-Creation property (Lemma 6.5) is deduced from the counting of messages on the receiver side while the 1-Reordering property (Lemma 6.6) comes from the fact that the delivering of a message by the receiver always happens during the execution of $\mathcal{SDL}$-**Send** by

the sender.

For the sake of the proof, let v_i and v_j be two neighboring vertices that execute $\mathcal{SDL}$, v_i being the sender and v_j the receiver. Let $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots$ be an execution starting from any arbitrary configuration γ_0.

We say that a message m' is processed by v_j when v_j executes **DeliverMessage**(m') (line 07 of $\mathcal{SDL}$-**Receive** function) if m' is a normal message or when v_j executes **DropMessage**(m') (line 09 of $\mathcal{SDL}$-**Receive** function) if m' is a $< SYNCHRO >$ message.

First, we need two preliminaries results related to the result of the execution of the procedure **AuxiliarySend** by v_i depending on the configuration in which v_i starts to execute this procedure.

Lemma 6.1

*When v_i starts to execute **AuxiliarySend**(m', ab) in a configuration where $ab \neq last_delivered$, the message m' (either a $< SYNCHRO >$ message or a normal message) and every message parameter to a subsequent invocation of **AuxiliarySend** is processed by v_j in a finite time.*

Proof: Consider a configuration γ_k where $ab \neq last_delivered$. Assume that v_i starts to execute **AuxiliarySend**(m', ab) in γ_k. By contradiction, assume m' is never processed by v_j in the remainder of σ. That is, v_j never executes lines 07 or 09 in the $\mathcal{SDL}$-**Receive** procedure. In turn, tests on lines 04 or 05 never evaluate to *true* simultaneously.

As $last_delivered \neq ab$ in γ_k and *last_delivered* may change only when m' is processed (line 10), we know that the test on line 05 is always *true* (since m is never processed by assumption).

This implies that $Q[m', ab] \geq c+1$ never evaluates to *true* (test on line 04). This implies that the sender stops sending (m', ab) before the (m', ab) counter reached $c+1$, which is impossible. The reason is as follows. In order to stop sending the same message, v_i must get $3c+2$ acknowledgments for the message $\{m', ab\}$. If such $3c+2$ acknowledgments are indeed received, this implies that the receiver issued at least $2c+2$ of those acknowledgments, and thus received $2c+2$ messages $\{m', ab\}$. Consider the first such message $\{m', ab\}$ received by v_j. If there is no reset of v_j's queue following this packet, the head of the queue now contains an entry $(m', ab, *)$ that can not be deleted until the receiver resets the entire queue. Indeed, at most c packets are initially present in the receiver's input channel, that can create at most c entries in the queue. Since the queue is of size $c+1$, the $(m', ab, *)$ tuple remains. Now, if the receiver sends $c+1$ acknowledgment for the message $\{m', ab\}$, it implies that the receiver's queue for entry $(m', ab, *)$ was incremented $c+1$ times, which invalidates the assumption. It follows that m' is processed in a finite time.

Note that after the processing of m', *ab* and *last_delivered* have the same value with the execution of the line 10 of $\mathcal{SDL}$-**Receive** procedure. Hence the next invocation of the **AuxiliarySend** primitive by v_i will make the values *ab* and *last_delivered* different. Applying the above reasoning, the lemma follows. ∎

Lemma 6.2

When v_i starts to execute **AuxiliarySend**(m', ab) *in a configuration where $ab = last_delivered$, only m' (either a $< SYNCHRO >$ message or a normal message) is not processed by v_j.*

Proof: Consider a configuration γ_k where $ab = last_delivered$. Assume that v_i starts to execute **AuxiliarySend**(m', ab) in γ_k.

Since the test in the line 05 of the $\mathcal{SDL}$**-Receive** procedure evaluates to *false*, the processing of m' is not executed. However, since v_i keeps sending m' and v_j acknowledges these packets the **AuxiliarySend** procedure returns. Note that v_i executes line 01 or 03 of the $\mathcal{SDL}$**-Send** procedure before the next invocation of **AuxiliarySend** procedure.

It follows that the system reaches in a finite time a configuration where $ab \neq last_delivered$. Then, Lemma 6.1 implies that every message that is parameter of subsequent invocations of **AuxiliarySend** is eventually processed by v_j. ■

Now, we can prove that $\mathcal{SDL}$ satisfies the four properties of the $(0,1,1,1)$-message performance (see Definition 6.1) starting from any configuration.

Lemma 6.3

$\mathcal{SDL}$ satisfies the 0-Loss property starting from any configuration.

Proof: Assume that v_i has to send a message m to v_j starting from an arbitrary configuration. Note that proofs of Lemmas 6.1 and 6.2 imply that any invocation of the $\mathcal{SDL}$**-Send** procedure eventually ends. This implies in turn that v_i starts to execute $\mathcal{SDL}$**-Send**(m) in a finite time.

Then, v_i invokes first **AuxiliarySend** with a $< SYNCHRO >$ message as parameter (see line 02 of the $\mathcal{SDL}$**-Send** procedure). Note that this $< SYNCHRO >$ message may be lost if $ab = last_delivered$ when v_i starts to execute **Auxiliary-Send** by Lemma 6.2.

Then, following Lemma 6.2, we have $ab \neq last_delivered$ when v_i starts to execute **AuxiliarySend** with m as parameter (see line 04 of the $\mathcal{SDL}$**-Send** procedure) since it has executed line 03 of the $\mathcal{SDL}$**-Send** procedure. By Lemma 6.1, it follows that m is eventually processed by v_j. As m is a normal message, this implies by definition that m is delivered to v_j in a finite time.

As this result holds whatever the state of the system when v_i requests to send m, we obtain that $\forall m \in S(\sigma), m \in R(\sigma)$. To conclude, observe that $S(\sigma) = S(\sigma)_0$. ■

Lemma 6.4

$\mathcal{SDL}$ satisfies the 1-Duplication property starting from any configuration.

Proof: By contradiction, assume that there exists an execution σ of $\mathcal{SDL}$ such that $\forall b \leq 1, \exists m \in S(\sigma), \left|\{m'_i = m | m'_i \in R(\sigma)\}\right| > 1 \wedge m \notin R(\sigma)^b$. In particular, this property is true for $b = 1$. Hence, $\exists m \in S(\sigma), \left|\{m'_i = m | m'_i \in R(\sigma)\}\right| > 1 \wedge m \notin R(\sigma)^1$. In other words, there exists in σ a message m sent by v_i delivered several times to v_j. Moreover m is not the first message received by v_j.

This implies that the line 07 in the $\mathcal{SDL}$-**Receive** procedure is executed several times for the message m. It is impossible and the reason is the following. After the first delivery of m the receiver empties the queue and makes *last_delivered* $= ab$ (see proof of Lemma 6.2). Note that v_i modifies ab only when it stops to send m. Even if v_i keeps invoking **SendMessage**(m, ab) until it receives the $3c+2$ acknowledgments, none of these messages will be delivered since for each of them the test in line 05 in the $\mathcal{SDL}$-**Receive** procedure returns *false*.

This contradiction implies that only the first message received by v_j may be duplicate. The lemma follows. ■

Lemma 6.5

$\mathcal{SDL}$ satisfies the 1-Creation property starting from any configuration.

Proof: By contradiction, assume that there exists an execution σ of $\mathcal{SDL}$ such that $\forall c \leq 1, \exists m \in R(\sigma), m \notin S(\sigma) \wedge m \notin R(\sigma)^c$. In particular, this property is true for $c = 1$. Hence, $\exists m \in S(\sigma), m \notin S(\sigma) \wedge m \notin R(\sigma)^1$. In other words, there exists in σ a message m not sent by v_i but delivered to v_j. Moreover m is not the first message received by v_j.

Initially the channel (i, j) may contain at most c ghosts messages. In the worst case, the receiver's queue also contains an entry for each of the ghost with the counters initialized to c or $c+1$.

Let $\{g, ab\}$ be the first ghost message received by v_j with alternated bit set to ab. Let us study the two possible cases. First, assume that $ab \neq$ *last_delivered*. Then v_j delivers g (line 07 of $\mathcal{SDL}$-**Receive** procedure) and empties the queue (line 11 of $\mathcal{SDL}$-**Receive** procedure). Second, assume that $ab =$ *last_delivered*. Then v_j does not deliver g (due to the test of line 05 of $\mathcal{SDL}$-**Receive** procedure) but it empties the queue (line 11 of $\mathcal{SDL}$-**Receive** procedure).

In both cases, there is at most one ghost message delivered to v_j and the queue has been emptied. In turn, it remains now at most $c-1$ ghosts messages in the channel (i, j). If one of them is received by v_j (after an invocation of **ReceiveMessage**), its associated counter cannot reach the value $c+1$ (unless v_i starts to send the same message but in this case, it is no longer a ghost message) since there are at most $c-1$ copies of the same message. Consequently, none of the $c-1$ remaining ghost messages can be delivered, that contradicts the construction of m and proves the result. ■

Lemma 6.6

$\mathcal{SDL}$ satisfies the 1-Reordering property starting from any configuration.

Proof: Following Lemma 6.5, $\mathcal{SDL}$ delivers at most one ghost message to v_j in σ. Let us consider the two following possible cases.

Case 1: $\mathcal{SDL}$ delivers no ghost message to v_j in σ.
According to Lemmas 6.3 and 6.4, any message sent from v_i is delivered to v_j exactly once in this case. Now, observe that any message is delivered to v_j between the beginning and the end of the corresponding execution of the procedure $\mathcal{SDL}$-**Send** by v_i. Indeed, the message is delivered to v_j when it receives the $(c+1)$-th copy of the message whereas v_i waits to receive the

$(3c + 2)$-th acknowledgment of the message to stop sending it (see proof of Lemmas 6.1 and 6.2). Since the $\mathcal{SDL}$**-Send** procedure is blocking for v_i, $R(\sigma)_0 = R(\sigma) = S(\sigma)$ for any execution σ where $\mathcal{SDL}$ delivers no ghost message to v_j. Hence, $\exists d = 0 \leq 1, R(\sigma)_d = S_\sigma$.

Case 2: $\mathcal{SDL}$ delivers one ghost message to v_j in σ.
Assume that the ghost message delivered by $\mathcal{SDL}$ is m. Lemma 6.5 allows us to state that m is the first message delivered to v_j. Then, a similar reasoning to the one of case 1 allows us to conclude that $R(\sigma) = m.S(\sigma)$ for any execution σ where $\mathcal{SDL}$ delivers one ghost message m to v_j and then, $R(\sigma)_1 = S(\sigma)$. Hence, $\exists d = 1 \leq 1, R(\sigma)_d = S(\sigma)$.

In both cases, we show that $\mathcal{SDL}$ satisfies the 1-Reordering property. ■

Now, we can conclude on the following corollary of Lemmas 6.3, 6.4, 6.5 and 6.6.

Theorem 6.4

$\mathcal{SDL}$ is a self-stabilizing distributed protocol for $spec_{DLC}$ over c-bounded channels (with v_i and v_j being respectively the sender and the receiver) with a $(0, 1, 1, 1)$-message performance.

6.2 Bounded Labelling Systems

This section aims to present a second necessary tool for our atomic register simulation provided in Chapter 7. This tool is a bounded labeling system, that is a set of labels (*a.k.a.* time-stamps) enhanced with a total antisymmetric binary relation (in order to compare labels with each other) that always allows the computation of a new label greater than (according to the binary relation) any bounded-sized set of labels.

Section 6.2.1 presents the problem and motivates the need of a new bounded labeling system to tolerate constraints of self-stabilizing settings and Section 6.2.2 presents our solution to this problem.

6.2.1 Problem and Related Works

In any system, it is often useful to enhance data or events with some information to compare them. For example, date and time of creation and of last modification are always associate to files in order to compare versions. In a centralized system, the value of the clock is a good information to keep trace of time. Due to asynchronism (each vertex has its own clock and its own speed), we cannot use clock values to date data or events in a distributed system since we are not able to compare clock values from two different vertices. The key idea is to store an information that allows to deduce what event precedes the other. Israeli and Li defined in [IL93] time-stamps as "numerical labels which enable a system to keep track of temporal precedence relation among its data elements".

In the following, we survey main results about time-stamping in distributed systems. We assume that we have some objects (data, messages, ...) to time-stamp in order to track temporal dependence between them. A natural way to construct such a time-stamps is to use the set of natural numbers with its total order. We associate to each object a natural number and when a vertex wants to time-stamp a new object, it scans all existing time-stamps and compute the maximum. Then, it remains to add 1 to obtain a new time-stamp that is greater than the one of all existing objects. In this way, objects may be ordered according to their time-stamps since the latest time-stamped object have the greatest time-stamp. The main drawback of this scheme is its unboundedness. Indeed, to remain correct, we must assume that time-stamps are unbounded that implies that each vertex needs an unbounded memory. This solution is simple but unsatisfactory.

Now, we describe a bounded solution to time-stamping provided by [IL93]. They provide a full specification of the problem in two elements: a set of labels and a labeling protocol. As the set of labels must be ordered by a total antisymmetric binary relation, we can represent it by a tournament (that is, a complete directed graph with no circuit of length 2). Israeli and Li models then a bounded time-stamp system (*a.k.a.* bounded labeling system) by a two-player game on this tournament. We say that a vertex of the tournament dominates another one when the arc between them is oriented from the second to the first. Objects of the system to time-stamp are represented by pebbles. A pebble is put on a vertex of the tournament when the corresponding time-stamp is associated to the object. We assume that there exists a bound k on the number of objects to label. Initially, pebbles are placed on pre-defined vertices of the tournament. The first player, the adversary, chooses an arbitrary pebble and the second player, the labeler, must move this pebble such that its new position dominates all others pebbles. Once this move is performed, the adversary chooses a new pebble and the game continues. A labeling protocol is a protocol that can supply the labeler whatever are the choices of the adversary. Choices of the adversary are assumed arbitrary to model any possible execution of the distributed system. The existence of such a tournament and protocol is not trivial.

As the adversary chooses only a pebble at a time, the bounded labeling scheme defined previously is sequential (that is, only one object can modify its time-stamp at a time). If we allow the adversary to choose several pebbles at a time, we must assume the existence of severals labelers and the bounded labeling system become concurrent [DS97].

As numerous distributed protocols need the existence of a bounded time-stamp system (see *e.g.* [Lam74, VA86]), the definition and the efficiency of such time-stamp systems is a challenging task. We now survey main results in this domain. As we previously mentioned, [IL93] was the first to introduce a definition of bounded time-stamp systems. It also provides the first sequential bounded time-stamp system that is constructible and efficiently computable (note that we describe it in details in the following). Israeli and Li also provide a concurrent bounded time-stamp system in this paper but this latter is not really usable in practice since its size is very far from

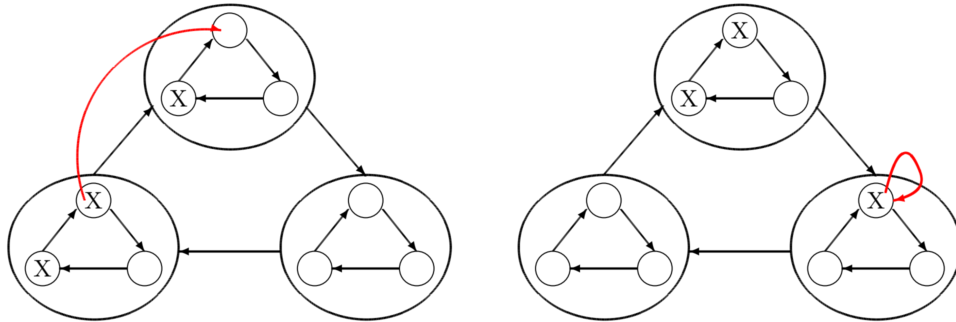

Figure 6.2: An example of the bounded labeling system of [IL93] of rank 3. An arrow from a set of vertices to another one indicates that any vertex of the first set is dominated by any vertex of the second. The red arrow indicates a possible pebble move.

the optimal value. Dolev and Shavit provide in [DS97] the first concurrent bounded time-stamp system that can be used in practice. This concurrent bounded time-stamp system was improved in [GLS92]. Note that all these bounded time-stamp systems need a correct initialization of the time-stamps (in other words, they are not suitable for a self-stabilizing context). To remedy to this fact, [Abr03] and [DKS10] propose self-stabilizing bounded time-stamp systems but they assume the atomic register model. By the way, these solutions are not convenient in our case since we want to simulate such a model in the message passing model using a bounded time-stamp system.

As our goal in this part is to provide a single-writer multi-reader atomic register simulation in presence of any transient and permanent crash fault pattern, we focus now on bounded sequential time-stamp systems that can cope with initial corruption of time-stamps. First, we describe the bounded sequential time-stamp system of [IL93] and we illustrate its inconvenience for our purposes. Then, we introduce the formal definition of a stabilizing bounded labeling system.

Israeli and Li define the associated tournament to their bounded sequential time-stamp system recursively. When there are only two objects to time-stamp, the tournament (of rank 2) is reduced to a circuit of size 3. When there are k objects to time-stamp, the tournament (of rank k) is the circuit of size 3 in which each vertex is substituted by the tournament of rank $k-1$. Figures 6.2 and 6.4 present examples of this tournament of rank 3 and 4 respectively. Once this tournament defined, it is quite easy to construct the labeling protocol. Figures 6.2 and 6.3 provide some examples of characteristic pebble moves on a time-stamp system of rank 3 (that is, with at most 3 pebbles). Given a distribution of the pebbles in the labeling system (depicted by crosses on the figures), the goal is to move the selected pebble to a new vertex such that this new position dominates all other pebbles (this move is depicted by the red arrow on the figures). Note that examples of these two figures are reachable from an initial configuration in which all pebbles are located at the same arbitrary vertex of the tournament. Indeed, if we consider an arbitrary

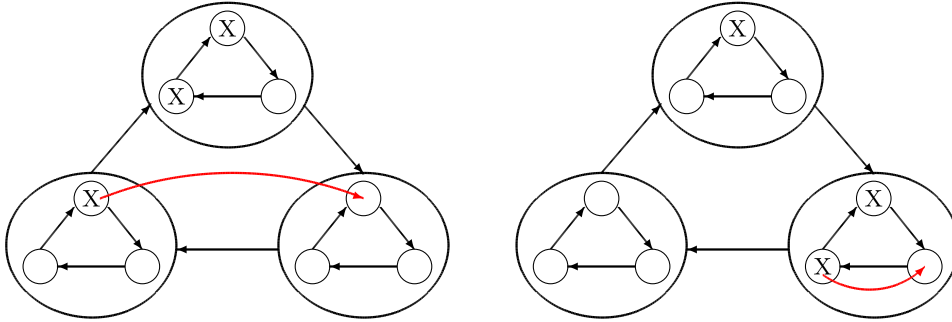

Figure 6.3: An example of the bounded labeling system of [IL93] of rank 3. An arrow from a set of vertices to another one indicates that any vertex of the first set is dominated by any vertex of the second. The red arrow indicates a possible pebble move.

initial location of pebbles, the labeling protocol may be unable to compute a new label. For instance, consider the example of Figure 6.4 (a tournament of rank 4 with 4 pebbles), we have 3 pebbles that form a circuit and the adversary wants the labeler to move the fourth, that is impossible since any vertex of the tournament is dominated by one of the three remaining pebbles.

To avoid such problems, we define a stabilizing bounded labeling system as a bounded labeling system with stronger properties. Indeed, we add the following constraint on the tournament: for any subset of at most k labels, there exists a label that dominates each label of the subset. In this way, we are ensured that a stabilizing bounded labeling system can deal with any arbitrary initialization since it is always possible to compute a label greater than the existing ones. We can define formally a stabilizing bounded labeling system in the following way:

Definition 6.2 (*Stabilizing bounded labeling system*)

A k-stabilizing bounded labeling system ($k \geq 2$) is a triplet $(L, \prec, next)$ where L is a finite set, $\prec$ is a total antisymmetric binary relation over L and $next$ is a function $next : L^k \to L$ such that:

$$\forall L' \subseteq L, |L'| \leq k \Rightarrow \forall \ell \in L', \ell \prec next(L')$$

Note that the definition of [IL93] is a particular case of ours since any stabilizing bounded labeling system is a bounded labeling system (by definition) while the converse is not true (see the counter-example described above).

6.2.2 Solution

The existence of a stabilizing bounded labeling system is a non trivial question. This question was raised (under another vocable) by Schütte in 1962 in graph theory. The first answer comes from Erdös that proves in [Erd63] by a probabilistic argument that such a tournament exists for any value of k. However, note that this solution is

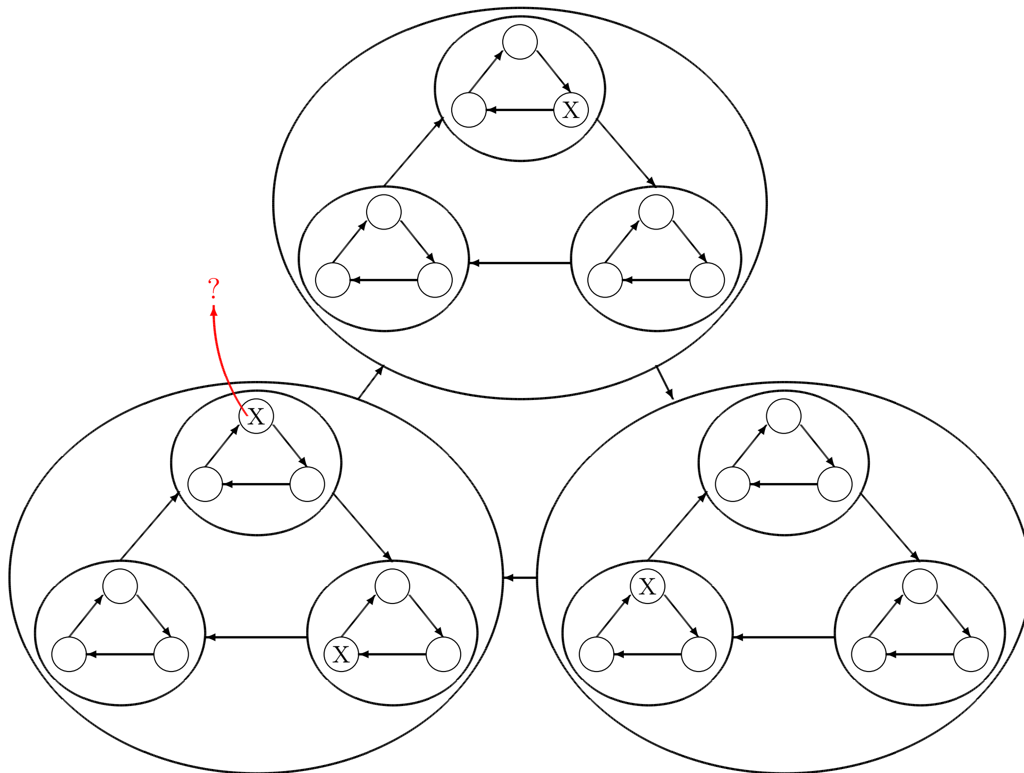

Figure 6.4: An example of the bounded labeling system of [IL93] of rank 4 in an arbitrary initial configuration (an arrow from a set of vertices to another one indicates that any vertex of the first set is dominated by any vertex of the second).

Protocol 6.4 Next: the labeling protocol of our stabilizing bounded labeling system.

Next

```
input:
k: a natural number
S = {(s_1, A_1), (s_2, A_2), ..., (s_k, A_k)}: set of k labels
output:
A label greater than any label of S
persistent variable:
ℓ = (s, A): a label
X: a set of natural numbers
function:
For any ∅ ≠ S ⊆ X, pick(S) returns an arbitrary element of S

01:  A := {s_1} ∪ {s_2} ∪ ... ∪ {s_k}
02:  X := {1, 2, ..., k^2 + 1}
03:  while |A| ≠ k do
04:     A := A ∪ {pick(X \ A)}
05:  s := pick(X \ (A_1 ∪ A_2 ∪ ... ∪ A_k))
06:  return (s, A)
```

not constructive. In [SS65], a lower bound of $(k+2)2^{k-1}-1$ on the number of vertices of this class of tournament was proved. Finally, [GS71] provides a constructive solution that is near to the optimal number of vertices. Nevertheless, this solution is too complex to be computed efficiently. Hence, we propose in the following a new stabilizing bounded labeling scheme that can be computed efficiently (at the price of a huge number of vertices). Note that the loss of the optimality with respect to number of vertices is not a real drawback since we do not store the tournament in memory.

We can now present our stabilizing bounded labeling system. Let $k > 1$ be a natural number (the rank of the tournament, that is the maximal number of objects to label), and let $K = k^2 + 1$. We consider the set $X = \{1, 2, .., K\}$ and let L (the set of labels) be the set of all ordered pairs (s, A) where $s \in X$, and $A \subseteq X$ has size k.

We define the following binary relation $\lhd$ among L:

$$(s_j, A_j) \lhd (s_i, A_i) \equiv (s_j \in A_i) \land (s_i \notin A_j)$$

Note that the binary relation $\lhd$ is antisymmetric by definition but is not total (for instance, two labels (s_j, A_j) and (s_i, A_i) are not comparable if $s_j \in A_i$ and $s_i \in A_j$). We define now the binary relation $\prec$ among L in the following way: if two elements of L are not comparable using $\lhd$, then we choose an arbitrary relation between them, otherwise we choose the relation defined by $\lhd$.

We define now a function $next : L^k \to L$ in the following way. Given a subset S of k labels $S = \{(s_1, A_1), (s_2, A_2), \ldots, (s_k, A_k)\}$, *next* returns a label (s, A) that satisfies:

- s is an element of X that is not in the union $A_1 \cup A_2 \cup \ldots \cup A_k$ (as the size of each A_i is k, the size of the union is at most k^2, and since X is of size $k^2 + 1$

such an s always exists).

– A is a subset of size k of X containing all values $(s_1, s_2, \ldots, s_k)$ (if they are not pairwise distinct, add arbitrary elements of X to get a set of size exactly k).

The pseudo-code of function *next* is provided in Protocol 6.4.

We can now prove our main result.

Theorem 6.5

$S_k = (L, \prec, next)$ is a k-stabilizing bounded labeling system.

Proof: The set L is of size $\binom{K}{k}K$ by definition. Hence, L is a finite set. By definition, $\prec$ is a total antisymmetric binary relation over L. Given any subset S of k labels from L, $(s, A) = next(S)$ satisfies: for any element (s_j, A_j) of S, $s_j \in A$ and $s \notin A_j$. By definition, we have $(s_j, A_j) \prec (s, A)$ for any (s_j, A_j) of S. According to Definition 6.2, we obtain the result. ■

Note that it is simple to compute $next(S)$ given a set S with k labels, and can be done in time linear in the total length of the labels given, *i.e.* in $O(k^2)$ time. This observation allows us to claim that our stabilizing bounded labeling scheme is efficient and can be used in practice. Finally, observe that, given a natural number k, any label of S_k needs $O(k \times log_2(k))$ bits to be stored.

Chapter 7

Atomic Register Simulation

Premature optimization is the root of all evil (or at least most of it) in programming.

Donald E. Knuth

Contents

7.1 The ABD Simulation **101**
7.2 The FTPS Simulation **103**
7.2.1 Distributed Protocol 104
7.2.2 Proof of Correctness 106
7.2.3 Conclusion 111

This chapter is the core of the second part of this thesis since we present our atomic register simulation. The contribution of this chapter is to prove that the classical ABD simulation can be turned into a fault-tolerant pseudo-stabilizing single-writer multi-reader atomic register simulation.

To perform this goal, recall that we need to use two tools defined in Chapter 6. The first one is a communication primitive (called data-link protocol) that ensures that the number of lost, ghost, duplicated, and re-ordered messages are bound by finite constants in any execution. In Section 6.1, we provide such a data-link protocol that ensures optimal bounds over our message passing model (defined in Section 2.3.1). Note that we can use any data-link protocol while it ensures the existence of finite bounds in any execution. The second tool is a stabilizing bounded labeling system that is composed from a finite set of labels and a total antisymmetric binary relation that allows comparison of labels. This labeling system has moreover the property that, given any bounded-sized set of labels, we can compute a new label greater than any label of this set. Note that this property does not imply the existence of a maximum label in any bounded-sized set of labels (it may exist circuits in this set). The labeling system defined in Section 6.2 fulfills these properties.

This chapter is organized as follows. In Section 7.1, we present the ABD simulation in details and we provide its fault-tolerant pseudo-stabilizing version in Section 7.2.

7.1 The ABD Simulation

This section aims to present in details the fault-tolerant single-writer multi-reader atomic register ABD simulation provided by Attiya, Bar-Noy, and Dolev in

[ABND95]. Their assumptions on the distributed system follow. They assume a complete identified (*i.e.* each vertex of the communication graph has a distinct identity, see Section 2.2) communication graph and an asynchronous distributed system subject to any (f, ℓ)-permanent crash fault pattern (with $2n > f$). Vertex w (also denoted v_0 for the sake of consistency in protocols) is the writer (that is, it can invoke both the write and the read operation) while vertices from v_1 to v_{n-1} are readers (that is, they can invoke the read operation only).

In the following, we present only the bounded ABD simulation (the unbounded version makes use of natural numbers to label values of the register and can be easily derived from the bounded version). In this simulation, we assume the existence of a sequential bounded labeling system (as the one of [IL93], see Section 6.2). Note that a concurrent bounded labeling system is not necessary since we design a single-writer register.

First, they define a communication primitive, called **Communicate**, that ensures the communication by quorum. This primitive broadcasts a given message to all vertices and waits until getting an acknowledgment for a majority of them (it is always possible since at most $\frac{n}{2} - 1$ vertices may crash in any execution). Note that this communication primitive is designed to deal with the properties of the considered message passing model (non reliable and non FIFO communication links). This communication by quorum is the basis of the distributed protocol and was extensively re-used.

In the following, we associate a label (from the sequential bounded labeling system) to each value of the register. Note that we ignore the actual value of the register from now and we consider only the management of the label for the sake of simplicity. As the labeling system is bounded, the writer must take into account all existing labels in the distributed system before computing a new one (indeed, the new label does not depend only of the writer label as in the unbounded version). The main difficulty is then to maintain the set of existing labels in the distributed system in spite of crashes and asynchrony. In other words, to ensure correctness, the writer must be aware of at least all existing labels in the distributed system when it computes a new one (note that the set of gathered labels may be greater and contains obsolete labels).

To reach this goal, the **Write** operation operates as follow. The writer collects (via the primitive **Communicate**) the existing labels in the distributed system (readers send labels that they have for the writer and the most recent labels that they have sent to other vertices). The writer computes then a new label greater than each label it collected. The problem is that the primitive **Communicate** ensures only the collect from a majority of vertices. In consequence, any correct vertex must ensure that its labels are stored at a majority (at least) of vertices at any time. In this way, the writer is able to gather all existing labels when it collects labels from any majority.

To this end, whenever a vertex adopts a new label (that it believes to be the maximum label of the writer), it invokes a procedure **Record** that stores this label and all the recent labels it has sent to other vertices using the primitive **Commu-**

nicate. A vertex receiving a recording message simply stores all the labels in its memory. In response to a query from the writer about labels, a reader sends all labels it has stored. This implies that no label may be lost (since a majority of vertices stores these labels). Note that, for avoiding chain reaction where a recording message causes other recording messages, vertices ignore the labels carried by recording messages even if their label is greater than their current writer label.

To implement this management of labels, the main data structure of the distributed protocol is L_i, an $n \times n$ matrix of labels, for each vertex v_i. The ith row $L_i[i]$ is updated dynamically by v_i according to messages it sends while other rows $L_i[j]$ $(j \neq i)$ are updated by messages that v_i received from v_j during **Record** invocations by v_j (that is, $L_i[j]$ is the latest view of v_i on $L_j[j]$). Each element $L_i[i,j]$ (for $j \neq i$), contains two fields: $L_i[i,j].sent$ and $L_i[i,j].ack$ that store respectively the last label that v_i sent to v_j and the last label acknowledged by v_j to v_i. Finally, the element $L_i[i,i]$ contains the current maximum label of the writer known to v_i.

The key idea of the correctness proof relies on the notion of viable label. A label ℓ is said to be viable and in the responsibility of a vertex v_i if either $L_i[i,i] = \ell$ (that is, v_i believes that the current register label is ℓ), $L_i[i,j].sent = \ell$, or $L_i[i,j].ack = \ell$ (that is, a non recording message containing ℓ has been sent from v_i to v_j). Now, a label is recorded if this label appears either in the writer matrix or in the matrices of at least a majority of vertices. The proof is based on the fact that any viable label is recorded in a finite time. Then, it is possible to show that, at each execution of the **Write** function, the new label is greater than any viable label in the distributed system. Then, the proof is analog to the one of the unbounded version.

In this way, the ABD simulation can tolerate any (f, ℓ)-permanent crash fault pattern (if $2n > f$). If we consider a distributed system subject to a transient and permanent crash fault pattern, then the ABD simulation may reach some undesirable configurations in which the bounded labeling system is unable to compute a new label greater than the existing ones. Even if we assume the existence of a stabilizing bounded labeling system to avoid such cases, the ABD simulation cannot deal with arbitrary initialization of labels since some initially corrupted labels may remain unknown to the writer but be included infinitely often in **Read** function decision sets. The goal of the following section is to adapt the ABD simulation in order to tolerate such initial memory corruptions.

7.2 The FTPS Simulation

This section aims to present a slight variant of the ABD simulation that can tolerate, in addition of permanent crash faults, any transient fault. We present a fault-tolerant pseudo-stabilizing single-writer multi-reader atomic register simulation over message passing model. As far we know, it is the first time that such a simulation is designed. First, we describe our distributed protocol in Section 7.2.1. We prove its correctness in Section 7.2.2 and provide its space complexity in Section 7.2.3.

Protocol 7.1 $\mathcal{PSARS}$: FTPS single-writer multi-reader atomic register simulation (read operation for any vertex v_i, write operation for the writer $w = v_0$ only).

Variables:
L_i: a matrix $n \times n$ with the following constraints:
- For any $j \neq k$, the element $L_i[j,k]$ contains two fields: $L_i[j,k].sent$ and $L_i[j,k].ack$. The first field is the last label that v_j sent to v_k in the last **Read** operation of v_j known at v_i. The second field contains the last label known at v_i sent by v_j to v_k when v_j replied to the v_k label request.
- For any j, the element $L_i[j,j]$ has two fields. The field $L_i[j,j].value$ provides information on the last label of the writer known by v_j. The second field $L_i[j,j].conflict$ gives information on a label that conflicts with the current label of a vertex and that may be not known at the writer.

$label_set_i$: a set of labels
Functions:
MaxLabel: returns the maximum label (according to $\prec$) of the label set supplied as parameter if it exists, $\perp$ otherwise
Next: returns a label greater than (according to $\prec$) any label of the set given as parameter
PickValue: returns an arbitrary element of any circuit (according to $\prec$) of the label set supplied as parameter if possible, $\perp$ otherwise

Read$_i$()

```
01: label_set_i :=ReadQuorum_i(read)
02: if MaxLabel(label_set_i) ≠ ⊥ then
03:   if L_i[i,i].value ≺MaxLabel(label_set_i)
04:     L_i[i,i].value :=MaxLabel(label_set_i)
05:     L_i[i,i].conflict := ⊥
06:     WriteQuorumPromote_i()
07:   WriteQuorumRecord_i()
08:   return L_i[i,i].value
09: else
10:   L_i[i,i].conflict :=PickValue(label_set_i)
11:   WriteQuorumRecord_i()
12:   return abort
```

Write$_0$()

```
01: label_set_0 := ReadQuorum_0(write)
02: L_0(0,0).value := Next(label_set_0)
03: WriteQuorumPromote_0()
```

7.2.1 Distributed Protocol

As we previously claimed, our distributed protocol is the pseudo-stabilizing version of the ABD simulation presented in details in Section 7.1. In this section, we explain first differences between our simulation and the ABD simulation. Then, we present formally our distributed protocol. Note that, for the sake of simplicity, we ignore the actual value of the register and we concentrate only on label associated to it (as in [ABND95]).

Recall that we assume an asynchronous distributed system subject to any (k, f, ℓ)-transient and permanent crash fault pattern (with $2n > f$). The communication graph is complete and identified. One vertex is distinguished to be the writer. We denote this vertex by $w = v_0$. Vertices from v_1 to v_{n-1} are readers. We assume the message passing model described in Section 2.3.3. We also assume that any pair of vertices are able to communicate using the data-link protocol defined in Section 6.1. More precisely, if a vertex v_i has a message m to send to v_j, it invokes $\mathcal{SDL}$-**Send**$_j(m)$. The data-link protocol delivers this message to v_j by invoking **DeliverMessage**$_i(m)$. Finally, we assume the existence of a stabilizing bounded labeling system as the one described in Section 6.2.2. This labeling system provides a set of labels L and two functions. The first one, **Next**, computes a label greater

than (according to $\prec$) any label of the set given as parameter. The second one, **MaxLabel**, returns the maximum label (according to $\prec$) of the label set supplied as parameter if this maximum exists, $\perp$ otherwise. Note that **MaxLabel** returns $\perp$ when there exists a circuit in the set of labels supplied as parameter (that is, there exists a subset of labels $\ell_0, \ldots, \ell_t$ such that $\ell_0 \prec \ell_1 \prec \ldots \prec \ell_t \prec \ell_0$).

Protocol 7.2 $\mathcal{PSARS}$: Auxiliary functions (for any vertex v_i).

Notations:
For any j, the notation $L_i[j]$ represents the jth row of the matrix L_i.
Variables:
$return_set_i$: a set of labels
$read_answer_i$: array of n booleans
$record_answer_i$: array of n booleans
$promote_answer_i$: array of n booleans

ReadQuorum$_i$(*type*)

```
01: read_answer_i := [0, 0, ..., 0]
02: read_answer_i[i] := 1
03: return_set_i := ∅
04: foreach j ∈ {0, ..., n − 1} \ {i} do
05:    SDL-Send_j(Inquiry(type))
06: while |{j, read_answer_i[j] = 1}| ≤ n/2 do
07:    wait
08: return (return_set_i)

upon DeliverMessage_j(Inquiry(type))
09: if type =' read' then
10:    SDL-Send_j(Answer_Read(L_i[i, i]))
11:    L_i[i, j].ack := L_i[i, i].value
12: else
13:    SDL-Send_j(Answer_Write(L_i))

upon DeliverMessage_j(Answer_Read(L_j[j, j])
14: L_i[j, j] := L_j[j, j]
15: read_answer_i[i] := 1
16: return_set_i := return_set_i ∪ L_i

upon DeliverMessage_j(Answer_Write(L_j))
17: L_i[j] := L_j[j]
18: read_answer_i[i] := 1
19: return_set_i := return_set_i ∪ L_i ∪ L_j
```

WriteQuorumPromote$_i$()

```
01: promote_answer_i := [0, 0, ..., 0]
02: promote_answer_i[i] := 1
03: foreach j ∈ {0, ..., n − 1} \ {i} do
04:    SDL-Send_j(Promote(L_i[i, i]))
05: while |{j, promote_answer_i[j] = 1}| ≤ n/2 do
06:    wait
07: foreach promote_answer_i[j] ≠ 0 do
08:    L_i[i, j].sent := L_i[i, i].value

upon DeliverMessage_j(Promote(L_j[j, j]))
10: if L_i[i, i].value ≺ L_j[j, j].value then
11:    L_i[i, i] := L_j[j, j]
12:    WriteQuorumRecord_i()
13:    SDL-Send_j(Ack_Promote())

upon DeliverMessage_j(Ack_Promote())
14: promote_answer_i[j] := 1
```

WriteQuorumRecord$_i$()

```
01: record_answer_i := [0, 0, ..., 0]
02: record_answer_i[i] := 1
03: foreach j ∈ {0, ..., n − 1} \ {i} do
04:    SDL-Send_j(Record(L_i[i])
05: while |{j, record_answer_i[j] = 1}| ≤ n/2 do
06:    wait

upon DeliverMessage_j(Record(L_j[j])
07: L_i[j] := L_j[j]
08: SDL-Send_j(Ack_Record())

upon DeliverMessage_j(Ack_Record())
09: record_answer_i[j] := 1
```

Our distributed protocol makes use of (almost) the same data structure as the ABD simulation. Each vertex v_i stores an $n \times n$ label matrix L_i. For any $j \neq k$, the element $L_i[j, k]$ contains the same fields as in the ABD simulation: $L_i[j, k].sent$ and $L_i[j, k].ack$. The only difference with the ABD simulation matrix is that, for any j, the element $L_i[j, j]$ contains now two fields: $L_i[j, j].value$ and $L_i[j, j].conflict$. The

field $L_i[j,j].value$ provides the last label of the writer known by v_j. In particular $L_i[i,i].value$ contains the last label of the writer that the v_i is aware. Note that this field is equivalent to the field $L_i[j,j]$ of the ABD simulation. The second field $L_i[j,j].conflict$ gives information on a label that conflicts with the current label of a vertex and that may be not known at the writer. This field is used to avoid that some initially corrupted labels remain unknown to the writer but is included infinitely often in **Read** function decision set.

Classically, our distributed protocol is composed of two primitives: **Read** (for any vertex) and **Write** (only for the writer v_0). When a reader v_i invokes its **Read** primitive, it collects first the labels of at least a majority of vertices and computes the maximum with **MaxLabel**. Two cases can appear:

1. **MaxLabel** returns a label. This value (if it exceeds the current label of the reader) is recorded in the distributed system in order to refresh the views of the other vertices on the last label of v_i. Note that, after the reception of this new value, a vertex updates the corresponding entry in its matrix. Vertex v_i finishes its **Read** operation by promoting its value in the distributed system. Upon the reception of the value to be promoted, the vertex v_j compares its current label with the label of the received value. If its local value is obsolete (the local label is less than the received label), then v_j adopts the new value and pushes it in the distributed system.
2. Whenever this maximum cannot be computed (in presence of a circuit in the set of collected labels), the **Read** operation aborts. That may happen if the distributed system is still in the stabilization phase and not all the hidden labels have been revealed. In this case, the reader changes its $L_i[i,i].conflict$ field to one of the labels that forms a circuit. The idea is to help in revealing all the hidden labels in the distributed system. Indeed, the conflicting value is then recorded in the matrices of a majority of vertices that prevents such conflicting values to infinitely often disturb **Read** operations without being considered by a **Write** operation. This case is the main difference between our simulation and the ABD one.

The **Write** operation is similar to the one of the ABD simulation. When the writer invokes this primitive, it first collects the latest labels in the distributed system (by asking any majority of vertices), then computes its next label using the **Next** function. Finally it starts a promotion of the new value in the distributed system.

Protocols 7.1 and 7.2 provide the formal implementation of our fault-tolerant pseudo-stabilizing single-writer multi-reader atomic register simulation called $\mathcal{PSARS}$ (for $\mathcal{P}$seudo-$\mathcal{S}$tabilizing $\mathcal{A}$tomic $\mathcal{R}$egister $\mathcal{S}$imulation).

7.2.2 Proof of Correctness

This section is devoted to the proof of the fault-tolerant pseudo-stabilization of $\mathcal{PSARS}$ for $spec_{ARS}$. According to properties of our data-link protocol proved in

Section 6.1 (in particular the fact that $R(\sigma) = S(\sigma)$ or $R(\sigma) = m.S(\sigma)$ where m is an arbitrary message for any execution σ), we know that any execution has an infinite suffix in which no ghost, duplicated or re-ordered messages are delivered (since there is only a finite number of communication links in the distributed system). We can conclude that any execution has an infinite suffix in which any delivered message was actually sent. For the sake of simplicity, we consider only such suffixes of executions in the sequel of this proof. Note that this assumption does not restrict the generality of the proof since we want to prove the pseudo-stabilization of our distributed protocol (that is, only the existence of an infinite suffix satisfying the specification, not the finiteness of a prefix that does not satisfy the specification).

Recall that the main difficulty in the proof of the ABD simulation is to show that, in each **Write** invocation, the label set supplied to **Next** contains at least all viable labels present in the distributed system. In our case, the difficulty on proving our atomic register simulation also comes from the presence of fake labels (due to the arbitrary initialization of matrices) in the distributed system that may disturb the good functioning of the distributed protocol.

In the following, we prove that the writer includes in its decision set (records) all the viable labels in the distributed system defined below. A label ℓ is said to be viable and in the responsibility of vertex v_i if it satisfies one of the following properties:

- $L_i[i,i].value = \ell$ or $L_i[i,i].conflict = \ell$
- $L_i[i,k].sent = \ell$ or $L_i[i,k].ack = \ell$
- there is a vertex v_j such that $L_j[i]$ contains ℓ in one of the fields *sent*, *ack*, *value* or *conflict*.

Note that our notion of viable label extends the one presented above and that any label in the distributed system is in the responsibility of a vertex. A viable label is recorded if this label is stored in the writer matrix or the matrix of any majority of vertices. In the following, we show that any label in the responsibility of a vertex eventually becomes recorded. Note that once a label is stored in the matrix of the writer or in the matrix of a majority of vertices, this label will be included in the computation of the new label of the writer and it will not generate new conflicts.

This observation motivates the following necessary assumption for the fault-tolerant pseudo-stabilization of $\mathcal{PSARS}$: if the writer crashes in an execution, then this crash happens after the first stabilized **Write** invocation (that is, a **Write** invocation during which the label set supplied to **Next** includes all the viable labels in the distributed system). In other words, an execution has an infinite suffix that satisfies $spec_{ARS}$ if the writer does not crash during this execution or if the writer crashes after the first stabilized **Write** invocation (we cannot provide any properties in the contrary case). In the sequel of this section, we consider only such executions.

Lemma 7.1

Any execution of $\mathcal{PSARS}$ has an infinite suffix where every ***Read*** *invocation does not abort.*

Proof: By contradiction, assume that there exists an execution of $\mathcal{PSARS}$ that has an

infinite suffix where **Read** invocations abort infinitely often. Since there is a finite number of vertices, there is a vertex v_i such that its **Read** invocations returns *abort* infinitely often. It follows that **MaxLabel** returns $\perp$ for the set of labels that v_i gathered via **ReadQuorum** invocation for each of these invocations. This situation can occur only when there is a label ℓ such that ℓ has not been written by the writer (a hidden label) or ℓ was introduced by the writer during the stabilization phase (that is, when the writer computed the next label, some of the labels in the distributed system had not yet been revealed) for each of these invocations.

Let r_1 be the first **Read** of v_i that aborts. Note that before returning v_i stores in its field $L_i[i,i].conflict$ a conflicting label ℓ_1 (that is, ℓ_1 appears in a circuit of the set of labels collected by r_1) and records this label in a majority of vertices. Consider the first **Write** operation w_1 that happens after r_1. By construction, w_1 starts with a **ReadQuorum** invocation. Since r_1 happens before, ℓ_1 has already been recorded in a majority of vertices. It follows that w_1 also retrieves ℓ_1 and includes it in the input set of its **Next** invocation. The new chosen label ℓ'_1 will be greater than ℓ_1. Then w_1 promotes its new label in a majority of vertices.

Consider a read operation r_2 of v_i that happens after w_1. By construction, r_2 collects labels from a majority of vertices. Two cases can appear.

1. The label ℓ'_1 is equals to the return value of **MaxLabel** on the collected set and v_i adopts then this label, promotes, and records it into a majority of vertices.
2. The vertex v_i cannot compute the maximum over the collected set. In this case, v_i picks a label in the collected set, ℓ_2, such that ℓ_2 is part of a circuit with respect to the $\prec$ relation. Then, v_i stores ℓ_2 in its $L_i[i,i].conflict$ field and records it into a majority of vertices.

Since the number of fake labels is finite, eventually all these labels will be learned by the writer and included in the input set for the computation of its next label. Let ℓ'_t be this label and let w_t be the **Write** operation that computes this label (hence the necessity of the assumption that at least one such **Write** operation happens). w_t promotes l'_t in the distributed system. Let r_t be the first **Read** invocation by v_i that happens after w_t. Then, v_i retrieves ℓ'_t in its read quorum set. By construction, ℓ'_t is greater than any label in the distributed system and hence adopted by v_i. We can repeat this reasoning to conclude that any **Read** invocation by v_i after r_t does not abort which contradicts the original assumption of the proof. It follows that in any execution there is an infinite suffix where each **Read** does not abort. ■

Lemma 7.2

Any execution of $\mathcal{PSARS}$ has an infinite suffix where, for any vertex, the labels in its responsibility become recorded either at the writer or in a majority, or stay forever out of the computation.

Proof: Let σ be an execution of $\mathcal{PSARS}$ starting from a configuration γ and let S be the set of labels in the responsibility of a correct vertex v_i in γ. Let ℓ be a label in S. We distinguish the following cases.

1. Vertex v_i is correct and executes at least one **Read** operation in σ. By definition, label ℓ can be in the responsibility of v_i for three reasons.

(a) Label ℓ is contained in one of the fields of $L_i(i,i)$. Let r be the first **Read** operation of vertex v_i. Then v_i either modifies its current label to ℓ', promotes the new label, and records it or keeps ℓ and records it. It follows that the label contained in $L_i[i,i].value$ is recorded in at least a majority of vertices in one of the fields $L_j[i]$ and replaces any previously recorded label. Hence, S becomes eventually either S with ℓ recorded by the writer or $S \setminus \{\ell\} \cup \{\ell'\}$ with ℓ' recorded by a majority of correct vertices.

(b) When ℓ is in a field $L_i(i,k)$, it will be replaced by another label at the first *Inquiry* message reception from v_j or the first execution of **Write-QuorumPromote** by v_i. If this never happens, ℓ remains out of the computation since ℓ will never be communicated.

(c) The matrix of some vertex v_j is corrupted in γ and contains the label ℓ in $L_j[i]$. If v_j is a correct vertex then v_j is eventually contacted by a **Write** operation and sends its whole matrix to the writer. Two cases may occur. Either label ℓ is recorded in the matrix of the writer and further used by the writer to compute the next value or ℓ is eventually replaced by a new value ℓ', new label of v_i (see previous case). Hence, S becomes eventually either S with ℓ recorded by the writer or $S \setminus \{\ell\} \cup \{\ell'\}$ with ℓ' recorded by a majority of correct vertices.

2. Vertex v_i is correct but never executes a **Read** operation in σ.

   Since v_i is correct then v_i eventually receives a promote message from the writer. Either v_i replaces ℓ by the new label of the writer, ℓ', or keeps ℓ. In both cases v_i records its current label in a majority of vertices. Hence either S stays unchanged and ℓ is recorded either in the writer or in a majority or S becomes $S \setminus \{\ell\} \cup \{\ell'\}$ and ℓ' is recorder either in the writer or in a majority.

3. Vertex v_i is crashed.

   If ℓ is locally recorded in v_i, then ℓ will never appear in σ since v_i does not reply to any **Write** or **Read** operations. Assume ℓ is stored at some correct vertex v_j in $L_j[i]$. Then, v_j eventually sends to the writer its matrix. Either ℓ is recorded at the writer or ℓ is forever ignored. Note that even if v_j responds to read requests issued from readers, ℓ will never be communicated.

In each case, the result holds. ■

In the following, a viable label will refer only to labels that do not stay forever out of the computation.

Lemma 7.3

Any execution of $\mathcal{PSARS}$ has an infinite suffix that satisfies the regularity property of $spec_{ARS}$.

Proof: Let σ be an infinite execution of $\mathcal{PSARS}$. Following Lemma 7.1 and Lemma 7.2, σ contains an infinite suffix, σ', where no **Read** invocation aborts and any **Write** operation includes in its decision set all the viable labels in the distributed system. By contradiction, assume there is a vertex v_i such that its **Read** invocations return an obsolete label infinitely often in σ.

That is, there exists a **Read** invocation r by v_i such that the label returned by r is either a fake label or a label corresponding to a previous write but not the most recent. In σ', r returns the output value of **MaxLabel** invoked over the set of labels returned by **ReadQuorum**.

Let w_1 and w_2 be two **Write** operations such that w_1 happens before w_2 and r. Since w_1 happens before r then the label computed by w_1 is promoted and recorded in at least a majority of vertices and is greater than any label in the distributed system. When r starts invoking **ReadQuorum** two cases may appear: (i) w_2 did not modify the writer label and did not start the promotion of the new label via **WriteQuorumPromote** or (ii) w_2 executed **WriteQuorumPromote**. In the first case, w_1's label is the largest label in the distributed system. When r invokes the **ReadQuorum**, it gets w_1's label (otherwise w_1 is not terminated) and returns this label. Hence, r cannot return a value older than the one written by w_1. In the second case, some vertices contacted during the **ReadQuorum** execution may send the w_1's label, other vertices the w_2's label. Since the label computed in w_2 is greater than the label computed in w_1, **MaxLabel** invoked in r will return w_2's label. Hence, r will return the last written value, that contradicts its construction. ■

Lemma 7.4

Any execution of $\mathcal{PSARS}$ has an infinite suffix that satisfies the no new/old inversion property of $spec_{ARS}$.

Proof: Let σ be an execution of $\mathcal{PSARS}$. Following Lemmas 7.1 and 7.3, σ has an infinite suffix, σ', that satisfies the regularity property of $spec_{ARS}$ and in which any **Read** invocation does not abort. In the following, we prove that σ' does not violate the new/old inversion property of $spec_{ARS}$.

Consider two **Write** operations w_1 and w_2 in σ' such that w_1 happens before w_2. Consider also two **Read** operations r_1 and r_2 such that r_1 happens before r_2 and w_1 happens before r_1 (following the transitivity of the relation "happens before", w_1 also happens before r_2). Assume that r_1 and r_2 are concurrent with w_2 and that a new/old inversion happens. That is, r_1 returns the label ℓ_2 written by w_2 and r_2 returns the label ℓ_1 written by w_1.

Since r_1 happens before r_2, then r_1 executes the following actions (before the start of r_2): it modifies its local label to ℓ_2, it also executes **WriteQuorumPromote** in order to help w_2 to push its label in the ditributed system and finally it executes **WriteQuorumRecord** in order to inform the distributed system on its new value. Since **WriteQuorumPromote** returns before r_1 finishes, then the label ℓ_2 is already adopted by at least a majority of vertices. That is, since $\ell_2 \succ \ell_1$ (w_1 happens before w_2), then ℓ_2 replaces ℓ_1 in the matrices of at least a majority of vertices and also a majority of vertices proceeds to the record of their new label.

We assumed r_2 returns ℓ_1. Since r_1 happens before r_2 then r_2 starts its **ReadQuorum** after r_1 returned, in particular after r_1 completed its **WriteQuorumPromote** operation. This implies that ℓ_2 is the label adopted by at least a majority of vertices and at least one vertex in this majority will respond while r_2 invokes its **ReadQuorum**. That is, r_2 collects at least one label ℓ_2 and since $\ell_2 \succ \ell_1$, r_2 should return this value. This contradicts the assumption r_2 returns ℓ_1. It follows that σ' satisfies the no new/old inversion property of $spec_{ARS}$. ■

7.2.3 Conclusion

This section concludes the Chapter 7 by providing a study of the memory complexity of our atomic register simulation. For the sake of the comparison, remind that the ABD simulation needs a total amount of memory on the distributed system in $O(n^5)$ bits (using the bounded labeling system of [IL93]).

Lemma 7.5

$\mathcal{PSARS}$ requires $O(n^5 \times log_2(n))$ bits per vertex. Consequently, the total amount of memory on the distributed system is in $O(n^6 \times log_2(n))$ bits.

Proof: Note that the set *label_set* which is the input of **Next** contains $2n^3$ labels. Hence, following Section 6.2.2, one label needs $O(n^3 \times log_2(n))$ bits to be stored. Since any vertex must store $2n^2$ labels, we have the result. ■

The previous results allow us to state the following theorem:

Theorem 7.1

*$\mathcal{PSARS}$ is a f-ftps distributed protocol for $spec_{ARS}$ provided that $2n > f$ and that the writer can crash only after its first stabilized **Write** invocation. It requires $O(n^6 log_2(n))$ bits of memory on the whole distributed system.*

Chapter 8

Conclusion of Part II

> People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.
>
> Donald E. Knuth

Contents

8.1 Summary of Contributions **113**
8.2 Concluding Remarks **114**

8.1 Summary of Contributions

The second part of this thesis focuses on computational model transformation. Indeed, it is simpler to design distributed protocols in high atomicity models than in low atomicity ones. For instance, the register model allows atomic read and write operations on some shared variables (called registers). Depending on the properties provided by these atomic operations, we can distinguish different classes of registers. We concentrate on the class of registers that provides the best possible property: atomicity (that is, any read operation returns the value of the register as if operations are sequential and not concurrent).

Simulation of atomic register over lower atomicity computational models was extensively studied using two (complementary) approaches:

– simulation from a weaker register model; and
– simulation from the message passing model.

At our knowledge, there existed no atomic register simulation tolerating both transient and permanent crash faults using the second approach. The main contribution of this part is to fill this gap. More precisely, our contribution is threefold.

Data-link protocol As we want to design a distributed protocol resilient to transient faults in our message passing model that allows communication links to be not reliable and non-FIFO, it is desirable to provide a communication black box (called data-link protocol) that ensures the best possible communication properties. Until now, existing self-stabilizing solutions to this problem did not consider non-FIFO communication channels and did not provide quantitative effects of the transient

faults on messages. In Section 6.1, we first introduced a set of definitions to remedy at this last point. We characterized the fault resiliency of a data-link protocol by the maximal number of lost, ghost, duplicated, and re-ordered messages that this protocol cannot avoid. Then, we provide a stabilizing data-link protocol that is optimal with respect to these four criteria. As this data-link protocol is designed as an independent communication black box, we claim that it may be useful by any other distributed protocol operating in our message passing model.

Stabilizing bounded labeling scheme A classical approach to keep track of temporal precedence between events in an asynchronous distributed system is to use labeling schemes (indeed, clock values are not suitable since we cannot compare clocks between vertices due to the asynchrony). A natural labeling system is the set of natural numbers but it leads to unbounded memory usage at each vertex. Hence, bounded labeling systems were extensively studied. All existing stabilizing bounded labeling systems are designed for the atomic register model. In Section 6.2, we proposed the first stabilizing bounded labeling system that makes no assumption about the computational model. As with the data-link protocol, this labeling scheme is defined independently of protocols that use it, making it re-usable in another context.

Atomic register simulation Chapter 7 is the core of this part. It focuses on our single-writer multi-reader atomic register simulation in distributed systems subject to any transient and permanent crash fault pattern. Our contribution is the following: we show that the classical ABD simulation (a single-writer multi-reader atomic register simulation in distributed systems subject to permanent crash fault pattern designed by Attiya, Bar-Noy and Dolev) can be turned into a fault-tolerant pseudo-stabilizing distributed protocol using our data-link protocol and our stabilizing bounded labeling scheme. It turns out that only a few changes are necessary once these two building bricks are incorporated in the initial distributed protocol.

8.2 Concluding Remarks

Results presented in this part show the possibility to use the atomic register model in the context of self-stabilization. Our main contribution is to simulate such a model directly from the message passing model (and not from a weaker register model as in previous works) in a distributed system subject to any transient and permanent crash fault pattern. Note that our message passing model is weak since it allows arbitrary message looses and re-ordering.

As our approach is modular, we can replace the data-link protocol or the stabilizing bounded labeling system. In this way, we can improve our atomic register simulation by providing a better data-link protocol (*e.g.* by improving its complexity in number of messages or in space) or a better stabilizing bounded labeling scheme (*e.g.* by improving its number of vertices or the time complexity of the

next function). Our results about atomic register simulation open new avenues of research. For instance, is it possible to automatically adapt the numerous existing fault-tolerant distributed protocols based on atomic register model to tolerate transient faults as well?

Our atomic register simulation is a pseudo-stabilizing version of the ABD simulation that is also fault-tolerant. An important open question is the following: is it possible to provide a fault-tolerant self-stabilizing atomic register simulation? We conjecture that the answer is negative (at least for solutions based on time-stamps) but we could not succeed to prove this impossibility result. Our intuition relies on the following observations. Due to the asynchrony and the communication using quorums (that is made necessary by the possibility of crashes), a reader may remain out of the computation for any arbitrary long time. Then, if this reader happens to have the greatest time-stamp when it joins the computation, other readers may adopt its incorrect value of the register, which violates the specification.

Part III

Unison

Chapter 9

Introduction of Part III

Time is a great teacher, but unfortunately it kills all its pupils.

Louis Hector Berlioz

Contents

9.1 Problem and Related Works **120**
 9.1.1 Problem 120
 9.1.2 Related Works 120
 9.1.3 Specification and Definitions 121
9.2 Contributions of Part III **124**
9.3 Fault-Tolerant Self-Stabilization **125**

The third part of this thesis focus on a classical problem of distributed systems: synchronization. Interest in synchronization comes from the following fact: in a fault-prone environment, stronger are the properties of synchronism of the distributed system easier is the design of a distributed protocol. This property is proved by fundamentals results of [FLP85]. Indeed, Fisher, Lynch and Patterson proved the impossibility of the consensus in an asynchronous permanent fault-prone distributed system whereas this problem is easily solvable in an asynchronous fault-free distributed system.

Therefore, a desirable way to design a distributed protocol is to design first a synchronizer (a distributed protocol that ensures some synchronization guarantees over an asynchronous system), to design a simpler distributed protocol for the initial problem using these synchronization guarantees, and then to compose these two distributed protocols to obtain a distributed protocol for the initial problem in an asynchronous environment.

More precisely, we consider in this part asynchronous unison [GH90, CFG92] that requires vertices to maintain synchrony between their counters called clocks. Specifically, each vertex has to increment its clock indefinitely while the clock drift from its neighbors should not exceed 1. Asynchronous unison is a fundamental building block for a number of principal tasks in distributed systems such as distributed snapshots [CL85] and synchronization [Awe85, AKM+07].

This chapter aims to present in details the asynchronous unison problem and some related works (see Section 9.1) and to summarize results presented in this part (see Section 9.2).

9.1 Problem and Related Works

This section focuses on a presentation of the asynchronous unison problem (see Section 9.1.1), on a survey of previous results (see Section 9.1.2) and finally to a formal specification of the problem studied in this part (see Section 9.1.3).

9.1.1 Problem

As we previously mentioned, the design of distributed protocols over synchronous distributed systems is easier than over asynchronous ones. Hence, it is desirable to develop a general scheme that allows the synchronization of asynchronous distributed systems. The key idea is to design a transformer that simulate a synchronous distributed system. Then, if we compose this transformer with a distributed protocol for synchronous distributed systems we obtain a distributed protocol running over an asynchronous distributed system. Such a transformer is called synchronizer [Awe85].

According to [RH90], the goal of a synchronizer is to generate pulsations on each vertex of the communication graph as these pulsations were generate by a global clock that each vertex can have the value immediately. This implies that all vertices have the same pulsation of the global clock at any given time. We speak then about strong synchronization. We can weaken the constraints on the global clock in the following way: we allow a vertex to start a new pulsation only if all its neighbors and itself are in the same pulsation. We speak then about weak synchronization.

There exists simple synchronizers (like the α-synchronizer of [Awe85]) in which pulsations are implicit (only determined by message exchanges). If the distributed system is incorrectly initialized (as in self-stabilization), it is possible that, in spite of the synchronization, the execution may never be in "phase". In order to solve this problem, a well-know scheme (see *e.g.* [Mis91, CFG92]) is to add to each vertex a counter, called clock, that stores the identifier of the current pulsation. A phase clock distributed protocol is a distributed protocol that ensures that clocks of the distributed system remain weakly synchronized and are infinitely often incremented.

We say that a distributed system is in unison when clocks of any pair of neighbors of the communication graph differ from at most one unit. In the following, we define an asynchronous unison as any phase clock distributed protocol (over an asynchronous system) that stabilizes to unison [GH90].

9.1.2 Related Works

In this section, we provide a short survey of previous works related to clock synchronization and unison. One key issue about unison protocols is the boundedness of the clock. Indeed, we mainly distinguish unbounded clocks (that is, the domain value of the clock is isomorphic to $\mathbb{N}$, the set of natural numbers with a total order) and bounded clocks (that is, the domain value of the clock is a finite set with a partial order). In the following, we also distinguish distributed protocols by their fault tolerance capacities.

Fault-free systems As the subject of this thesis is related to fault-tolerance, we only cite the seminal work of Misra [Mis91] that clearly defined the problem of strong phase synchronization. Note that the communication graph is complete and that self-stabilization is not discussed. However, bounded clocks are considered.

Self-stabilizing systems The first work about unison (considering our definition) is from Gouda and Herman [GH90] that deals with synchronous systems and considers only unbounded clocks. The next step is achieved by Couvreur, Francez and Gouda that propose in [CFG92] an asynchronous unison based on bounded clocks. Considering specific communication graphs, some other works deal with asynchronous unison with bounded clocks and improve some characteristics of the general solution (see *e.g.* [HG95, LS95, HL98, LH01, BPV06]).

Finally, Boulinier, Petit, and Villain propose in [BPV04] a new unison that is optimal in term of number of states of the clock. Indeed, they prove a tight lower bound of $cycle(g) + hole(g) - 1$ states per clock vertex (where $cycle(g)$ is the length of the maximal cycle of the shortest maximum cycle basis of g and $hole(g)$ is the length of the longest chordless cycle of g). On the other side, they study optimality of states number in synchronous unison in [BPV05].

Multi-tolerant systems At our knowledge, clock synchronization in systems subject simultaneously to transient and permanent failures was only studied in synchronous settings.

First, we can cite probabilistic solutions designed for complete communication graphs. In this way of research, Dolev and Welch design in [DW04] an unison that support up to a third of Byzantine vertices but that has an exponential expected stabilization time. Ben-Or, Dolev and Hoch improve the stabilization time in [BODH08] since they provide a solution with a constant expected stabilization time under the same assumptions.

Then, if we consider deterministic solutions, we can distinguish those running only on complete communication graphs. Results of [PT97] and [DW97] focus on wait-free clock synchronization, that is they consider systems subject to transient and intermittent crash fault patterns while [HDD06] and [DH07] consider systems subject to transient and permanent Byzantine fault patterns. These two last works have a linear convergence time. The first tolerates up to a fourth of Byzantine vertices whereas the second tolerates up to a third of Byzantine vertices.

Finally, Dolev studied the strong clock synchronization in systems subject to transient and permanent crash fault pattern in [Dol97]. This work focuses on general communication graph and uses a bounded clock.

9.1.3 Specification and Definitions

Specification In the following, c_v is the variable of vertex v that represents its clock value. Values are taken in the set of natural integers (that is, the number of states is unbounded, and a total order can be defined on clock values). Note that

we do not consider the case of bounded clocks in this thesis. We now define a notion related to local clock synchronization. We call clock drift between two neighbors v and u the absolute value of the difference between their clock values. In this part, we deal with unison that is a weak clock synchronization: we must ensure that clocks are eventually "close" from each other. More precisely, two neighbors v and u are in unison if the drift between them is no more than 1. We say that a configuration of the communication graph is weakly synchronized if any correct vertex is in unison with its correct neighbors.

Definition 9.1 (*Weakly synchronized configuration*)

A configuration $\gamma \in \Gamma$ is weakly synchronized (denoted $\gamma \in \Gamma_1$) if and only if :

$$\forall v \in V\, \forall u \in N_v, u \text{ and } v \text{ corrects in } \gamma \Rightarrow |c_v - c_u| \leq 1$$

Intuitively, classical asynchronous unison ensures that the distributed system is eventually (and remains forever) in a weakly synchronized configuration (safety property) and that clocks of correct vertices are infinitely often modified (liveness condition). The only allowed modification of clocks after stabilization is incrementation. More formally, the classical specification of asynchronous unison [GH90, CFG92] follows.

Specification 9.1 (*Classical asynchronous unison $spec_{CAU}$*)

An execution σ satisfies $spec_{CAU}$ if and only if it complies with the following two properties:

(Safety) *every configuration of σ is weakly synchronized; and*

(Liveness) *the clock of every correct vertex is incremented infinitely often and never decremented in σ.*

Unfortunately, in the context of executions subject to transient and intermittent Byzantine fault pattern, we have the following result.

Proposition 9.1

There does not exist a strictly stabilizing distributed protocol for $spec_{CAU}$ (even when there is only one Byzantine vertex) for any containment radius.

Informal argument follows. Consider the following initial configuration: the Byzantine vertex b has a clock value of 0 and any correct vertex has a clock value equal to the distance between it and b. Then, this configuration satisfies the safety requirement of $spec_{CAU}$. Assume now that the Byzantine vertex takes no actions and keeps its clock value to 0. Remember that asynchronism of the system implies that this execution is indistinguishable from the one where b is a correct vertex and is very slow. Consequently, no correct vertex can increment its clock without violating the safety requirement of $spec_{CAU}$ from this configuration. Hence, no correct vertex can increment its clock infinitely often in any run starting from this configuration.

To make the problem solvable in the context of this thesis (executions subject to any transient and intermittent Byzantine fault pattern), we weaken $spec_{CAU}$ to obtain the specification of asynchronous unison as follows.

Specification 9.2 (*Asynchronous unison* $spec_{AU}$)

An execution σ satisfies $spec_{AU}$ if and only if it complies with the following two properties:

(Safety) *every configuration of σ is weakly synchronized; and*

(Liveness) *the clock of every correct vertex is incremented infinitely often in σ.*

This specification is weaker than $spec_{CAU}$ since it allows both increments and decrements (after clock synchronization) as long as the vertices remain in synchrony.

At this step, one may think about a very simple distributed protocol that may seem to be strictly stabilizing for $spec_{AU}$. The idea of this distributed protocol is to allow clocks of correct vertices to cycle between the values 0 and 1 whatever is the clock of the Byzantine vertices. This distributed protocol is composed of two rules. The first one sets the clock value of the vertex to 1 when its value is 0. The second one sets the clock value of the vertex to 0 when its value is not 0. This simple solution ensures our liveness property but is not strictly stabilizing since the closure of the safety property is not guaranteed. Consider the following counter-example: in the initial configuration of the distributed system, any clock has the same value (strictly greater than 2), say 15 for example and there is no Byzantine vertex (remember that clock values are unbounded integers by specification). Note that this configuration satisfies the safety condition of $spec_{CAU}$ (the clock drift between any two correct neighbors is at most one). Then, any correct vertex is enabled by the proposed distributed protocol. Assume now that the daemon chooses only one correct vertex, the next configuration does not satisfy the safety condition of $spec_{CAU}$ (since the chosen vertex takes the clock value 0 whereas its correct neighbors keep the clock value 15).

Minimality and priority We now present two key properties satisfied by all known self-stabilizing unison distributed protocols. Those properties are used in the impossibility results presented in Chapter 10. We called these properties respectively minimality and priority.

Minimality means that vertices maintain no extra variables but the digital clock value. This implies that a minimal distributed protocol for $spec_{AU}$ can only refer to clocks or to predefined constants. We now state the formal definition of this property.

Definition 9.2 (*Minimality*)

A distributed protocol for $spec_{AU}$ is minimal if and only if every vertex only maintains a clock variable.

Priority means that if, for a given vertex, incrementing the clock value does not break the local safety predicate with its neighbors, then its clock value is actually incremented in a finite number of activations of this vertex, even if no neighbor modifies its clock value. This property implies that, if a vertex can increment its clock without breaking unison with its neighbors, then it does so in finite time whether its neighbors are Byzantine, crashed or correct. This property is similar to obstruction-freedom [HLM03] in the sense that the distributed protocol only has very weak constraints about progress. We formally state this property in the following definition.

Definition 9.3 (*Priority*)

A distributed protocol for $spec_{AU}$ is priority if and only if it satisfies the following property: if there exists a vertex v such that for any $u \in N_v$, ($c_u = c_v$ or $c_u = c_v + 1$) holds in a configuration γ_i, then there exists a portion of execution $\sigma = (\gamma_i, \gamma_{i+1})...(\gamma_{i+k-1}, \gamma_{i+k})$ such that:

- *only v is chosen by the daemon during σ;*
- *c_v is not modified during actions $(\gamma_{i+j}, \gamma_{i+j+1})$, for $j \in \{0, ..., k-2\}$; and*
- *c_v is incremented during action $(\gamma_{i+k-1}, \gamma_{i+k})$.*

For example, distributed protocols proposed by [BPV04, BPV05, CFG92, GH90] fall in the category of minimal and priority unison using these definitions. Another example is the protocol of [PT97] that is priority but not minimal. To our knowledge, any existing unison distributed protocol satisfies either minimality or priority.

9.2 Contributions of Part III

Position of Part III As highlighted by related works described in Section 9.1.2, there does not exist, at our knowledge, any solution to clock synchronization problems for asynchronous distributed systems subject to composite fault patterns. The main goal of the Part III of this thesis is to fill this gap by studying asynchronous unison in distributed systems subject to transient and intermittent Byzantine fault patterns.

Moreover, we focus only on deterministic solutions on general anonymous communication graphs. Note that we consider only state model computational model in this part (see Section 2.3.2).

Overview of Part III The first contribution of this part is the generalization of fault-tolerant self-stabilization by the introduction of the idea of containment radius as in strict-stabilization (see Section 4.2.3). We say that a distributed protocol is (f, r)-fault-tolerant and self-stabilizing if, starting from any arbitrary configuration, every execution reaches a configuration from which the effect of a maximum of f crash faults is contained to the r-neighborhood of crashed vertices (see Section 9.3 for a formal definition). Then, contributions of this part are the following.

Impossibility results: Strong assumptions made on the distributed system and on the problem obviously lead to a couple of impossibility results. Chapter 10 details them. We prove impossibility results for FTSS asynchronous unison (that naturally imply similar impossibility results for strictly stabilizing asynchronous unison, see Section 9.3) related to the number of crashed vertices, the fairness of the daemon, the considered communication graph, and/or the minimality or the priority of the distributed protocol.

Possibility results: Chapter 11 brings a strictly stabilizing solution to asynchronous unison in the remaining possible cases. Moreover, we prove that this solution has an optimal stabilization time.

Finally, Chapter 12 concludes Part III by proposing a generalization of asynchronous unison and exposing some open questions.

Results presented in Chapter 10 have been published in Theoretical Computer Science [DPBT11] and in the proceedings of the 23rd International Symposium on Distributed Computing (DISC 2009) [DPBT09]. Chapter 11 leads to a publication in the proceedings of the 14th International Conference On Principles Of DIstributed Systems (OPODIS 2010) [DPBNT10].

9.3 Fault-Tolerant Self-Stabilization

Definition of fault-tolerant self-stabilization introduced in Section 4.2.1 states that a f-ftss distributed protocol is self-stabilizing in spite of f crashed vertices. This definition can be easily generalized using the key idea of containment radius of strict-stabilization (see Section 4.2.3).

Given a containment radius r, a distributed protocol is (f, r)-ftss if it ensures that, starting from any arbitrary configuration, any execution subject to at most f crashed faults reaches in a finite time a configuration from which any correct vertex that has no crashed vertice in its r-neighborhood satisfies the specification.

The formal definition follows. For any natural number r, we define g_r as the communication subgraph of g induced by the following set V_r (where C denotes the set of crashed vertices):

$$V_r = \{v \in V | \min_{c \in C}\{dist(g, v, c)\} > r\}$$

Definition 9.4 (*Fault-tolerant self-stabilization*)

A distributed protocol π is (f, r)-fault-tolerant and self-stabilizing ((f, r)-ftss for short) for specification spec if and only if starting from any arbitrary configuration every execution of π involving at most f crashed vertices contains a configuration from which every execution σ of π satisfies: the projection of σ on g_r satisfies spec.

From this definition, it could be easily deduced that impossibility results of Chapter 10 related to (f, r)-ftss distributed protocols imply similar impossibility

results about (c, f)-strict stabilization.

Chapter 10

Impossibility Results

> The limits of the possible can only be defined by going beyond them into the impossible.
>
> Arthur C. Clarke

Contents

10.1 General Results **129**
10.1.1 Two and more Byzantine Faults 129
10.1.2 Unfair Daemon 129
10.2 Minimal Unison Related Results **130**
10.2.1 Weakly Fair Daemon 130
10.2.2 Strongly Fair Daemon and Maximal Degree greater than 3 . 133
10.3 Priority Unison Related Results **136**
10.3.1 Weakly Fair Daemon 137
10.3.2 Strongly Fair Daemon and Maximal Degree greater than 3 . 138
10.4 Summary of Impossibility Results **140**

In this chapter we present a broad class of impossibility results related to the strictly stabilizing asynchronous unison. These impossibility results focus on the number of faulty vertices, on the topology of the communication graph, on the fairness of the daemon, or on the distributed protocol itself (minimality or priority of the distributed protocol). The extent of these impossibility results shows that it is practically vain to solve the unison problem in presence of any transient and intermittent Byzantine fault pattern. Note that Chapter 11 provides a strictly stabilizing distributed protocol to asynchronous unison when possible.

Note that all impossibility results of this chapter are actually related to FTSS distributed protocols, that obviously prove the impossibility of strictly stabilizing distributed protocols (see Chapter 4). Moreover, all impossibility results of this chapter are proved under central daemons for the sake of generality (since such an impossibility result implies impossibility for any other distribution of the daemon, see Corollary 3.2) whereas no assumptions are made on the boundedness or the enabledness of the daemon.

Details on the contribution of this chapter follow. First, we show a preliminary result that states that a vertex v cannot modify its clock value if it has two neighbors u and u' with $c_u = c_v - 1$ and $c_{u'} = c_v + 1$ (Lemma 10.1). This property is further used in the sequel of this chapter. Theorem 10.1 proves that there exists no (f, r)-ftss

distributed protocol for any r value if $f \geq 2$. Furthermore, in Theorem 10.2, we prove that there exists no $(1,r)$-ftss distributed protocol for $spec_{AU}$ under the central unfair daemon for any r value. Then we study the minimal and priority asynchronous unison and prove there exists no minimal or priority $(1,r)$-ftss distributed protocol for $spec_{AU}$ under the central weakly fair daemon for any r value (Lemma 10.2, Theorems 10.3 and 10.5). Finally, we prove there exists no minimal or priority $(1,r)$-ftss distributed protocol for $spec_{AU}$ under the central strongly fair daemon for any r value if the communication graph has a maximal degree of at least 3 (Lemma 10.3, Theorems 10.4 and 10.6).

Preliminary Result We introduce a preliminary result that shows that in any execution of a (f,r)-ftss distributed protocol for $spec_{AU}$ (under any central daemon) a vertex v cannot modify its clock value if it has two neighbors u and u' such that: $c_u = c_v - 1$ and $c_{u'} = c_v + 1$. This result follows simply from the fact that in the contrary case, the closure of $spec_{AU}$ is not satisfied. This result is extensively used in the sequel of this chapter.

Lemma 10.1

Let π be a (f,r)-ftss distributed protocol for $spec_{AU}$ (under any central daemon) and γ be a configuration where a vertex v (such that $c_v \geq 1$) has two neighbors u and u' such that: $c_u = c_v - 1$ and $c_{u'} = c_v + 1$. If v executes an action of π during an action (γ,γ'), then this action does not modify the value of c_v. If π is also minimal, then the vertex v is not enabled by π in γ.

Proof: Let π be a (f,r)-ftss distributed protocol for $spec_{AU}$ (under any central daemon). Let g be a communication graph and γ be a configuration of g such that no vertex is crashed, $\gamma \in \Gamma_1$, and there exists a vertex v (such that $c_v \geq 1$) with two neighbors u and u' such that: $c_u = c_v - 1$ and $c_{u'} = c_v + 1$.

Assume that v is activated by the daemon during an action (γ,γ') (only v is activated during this action since the daemon is central) and that this action modifies the value of c_v. Note that c_u and $c_{u'}$ are identical in γ and γ'. Let c be the value of c_v in γ and c' be the value of c_v in γ'. Values of c and c' satisfy one of the two following relations:

Case 1: $c < c'$.
This implies that $|c' - c_q| = |c' - c| + |c - c_q| > 1$ (since $|c' - c| \geq 1$ by hypothesis and $|c - c_q| = 1$).

Case 2: $c' < c$.
This implies that $|c' - c_{q'}| = |c' - c| + |c - c_{q'}| > 1$ (since $|c' - c| \geq 1$ by hypothesis and $|c - c_{q'}| = 1$).

In the two above cases, $\gamma' \notin \Gamma_1$, hence the closure property of π is not satisfied, that is contradictory. If π is also minimal, then the previous result implies that v is not enabled by π in γ. ■

10.1 General Results

This section proposes two impossibility results related to the number of permanently crashed vertices (see Section 10.1.1) and to the unfairness of the daemon (see Section 10.1.2).

10.1.1 Two and more Byzantine Faults

We prove here that it is impossible to provide a FTSS distributed protocol for asynchronous unison if there is at least two permanently crashed vertices. This result relies on the following observation: for any $(2, r)$-ftss distributed protocol for $spec_{AU}$, two crashed vertices at the extremities of a chain of length $2r+3$ may starve all correct vertices.

Theorem 10.1

For any natural number r, there exists no (f, r)-ftss distributed protocol for $spec_{AU}$ under any central daemon if $f \geq 2$.

Proof: Let r be a natural number. Let π be a $(2, r)$-ftss distributed protocol for $spec_{AU}$ (under any central daemon). Consider the following communication graph $g = (V, E)$ with $V = \{v_0, \ldots, v_{2(r+1)}\}$ and $E = \{\{v_i, v_{i+1}\} | i \in \{0, \ldots, 2r+1\}\}$ (that is, g is reduced to a chain of $2r + 3$ vertices). Let γ be the following configuration of g: v_0 and $v_{2(r+1)}$ are crashed and $\forall i \in \{0, \ldots, 2(r + 1)\}, c_{v_i} = i$ (all the other variables may have any value).

By Lemma 10.1, no vertex between v_2 and v_{2r+1} can change its clock value in every execution starting from γ. This contradicts the definition of π. Indeed, v_{r+1} must eventually satisfy the specification $spec_{AU}$ since the closest crashed vertex is at r hops away. In particular, any execution starting from γ must contain a suffix where the clock of v_{r+1} is infinitely often incremented. This contradiction shows us the result. ■

10.1.2 Unfair Daemon

This section is devoted to the following impossibility result: even with a single crashed vertex, it is impossible to provide a FTSS distributed protocol for asynchronous unison under the central unfair daemon. This result is deduced from the fact that, in any fault-free configuration, there exists at least two enabled vertices (otherwise, there is starvation in the case where a vertex is crashed). In this way, the unfair daemon may always starve a given vertex that leads to the result.

Theorem 10.2

For any natural number r, there exists no $(1, r)$-ftss distributed protocol for $spec_{AU}$ under the central unfair daemon.

Proof: Let r be a natural number. Assume that there exists a $(1, r)$-ftss distributed protocol π for $spec_{AU}$ under the central unfair daemon. Consider a communication

graph g, of diameter greater than $2r+2$ (note that in this case, at least one vertex must eventually satisfy the specification $spec_{AU}$). Let v be a vertex of V. Since the daemon is unfair, it can choose to never activate v in an execution σ unless this vertex becomes the only enabled vertex of g in a configuration of σ by definition.

For the sake of contradiction, assume that there exists a configuration γ such that no vertex is crashed and where v is the only enabled vertex of the communication graph. Denote by γ' the same configuration where v is crashed. Note that the set of enabled vertices is identical in γ and γ' by construction. As we assumed that only v is enabled in γ, this implies that no correct vertex is enabled in γ'. Hence, the system is deadlocked in γ' and the specification of $spec_{AU}$ is not satisfied since no clock of correct vertex can be updated. This contradiction implies that, for any configuration where no vertex is crashed, at least two vertices are enabled.

Since there exists no configuration where v is the unique enabled vertex (in every execution starting from any arbitrary configuration), the unfair daemon can starve v infinitely (if no crash occurs). This contradicts the liveness property of $spec_{AU}$ guaranteed by π since v cannot update its clock in this execution. ■

10.2 Minimal Unison Related Results

In this section, we prove two impossibility results related to minimal asynchronous unison, namely that it is impossible to provide a minimal FTSS distributed protocol for $spec_{AU}$ when the daemon is weakly fair (see Section 10.2.1) or when the daemon is strongly fair and the maximal degree of the communication graph is greater than 3 (see Section 10.2.2).

10.2.1 Weakly Fair Daemon

In this section we prove there exists no minimal $(1,r)$-ftss distributed protocol for $spec_{AU}$ under the central weakly fair daemon for any r value.

The impossibility result uses the following property: if there exists a minimal distributed protocol π that is $(1,r)$-ftss for $spec_{AU}$ under the central weakly fair daemon for a natural number r, then an arbitrary vertex v is not enabled by π if it has only one neighbor v' and if $c_v = c_{v'}$ (proved in Lemma 10.2 formally stated below). Then, we show that π starves the communication graph reduced to a two-correct-vertex chain where all clock values are identical (see Theorem 10.3).

Lemma 10.2

If there exists a minimal distributed protocol π that is $(1,r)$-ftss for $spec_{AU}$ under the central weakly fair daemon for a natural number r, then an arbitrary vertex v is not enabled by π if it has only one neighbor v' and if $c_v = c_{v'}$.

Proof: Let r be a natural number. Let π be a minimal $(1,r)$-ftss distributed protocol for $spec_{AU}$ under the central weakly fair daemon.

Let g be the communication graph reduced to a chain of length $r+2$. Assume vertices of g are labeled from left to right as follows: $v_0, v_1, \ldots, v_{r+2}$. Consider the following configurations of v (see Figure 10.1):

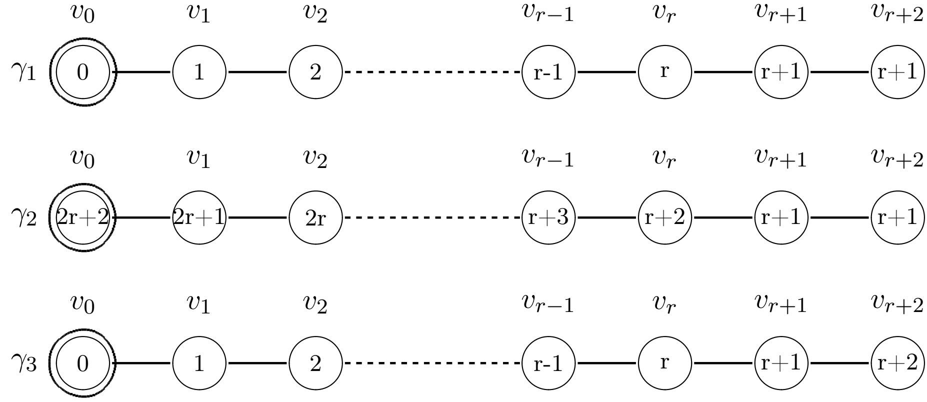

Figure 10.1: The three configurations used in the proof of Lemma 10.2 (the numbers represent clock values and the double circles represent crashed vertices).

- γ_1 defined by $\forall i \in \{0, \ldots, r+1\}, c_{v_i} = i$ and $c_{v_{r+2}} = r+1$ and v_0 crashed.
- γ_2 defined by $\forall i \in \{0, \ldots, r+1\}, c_{v_i} = 2r+2-i$ and $c_{v_{r+2}} = r+1$ and v_0 crashed.
- γ_3 defined by $\forall i \in \{0, \ldots, r+2\}, c_{v_i} = i$ and v_0 crashed.

By Lemma 10.1, vertices from v_1 to v_r are not enabled in such configurations (and remain not enabled until one of the vertices within $v_0, \ldots, v_{r+1}$ executes a rule).

Note that for the vertex v_{r+2}, the configurations γ_1 and γ_2 are indistinguishable (otherwise the unison would not be minimal). We are going to prove the result by contradiction. Assume that v_{r+2} is enabled in γ_1 and γ_2. The closure property of π implies that the enabled rule for v_{r+2} modifies its clock either to $r+2$ or to r. In the following we discuss these cases separately:

Case 1: The enabled rule for v_{r+2} modifies its clock into $r+2$.
As the daemon is central, v_{r+2} is the only activated vertex. Hence its clock takes the value $r+2$. The following cases are possible in the obtained configuration:

Case 1.1: v_{r+2} is not enabled.
If an execution started from γ_1, then no vertex is enabled in any configuration of this execution, that contradicts the liveness property of π.

Case 1.2 : v_{r+2} is enabled and the enabled rule modifies its clock into $r+1$.
Let σ be an execution starting from γ_1 where only v_{r+2} is activated. Consequently, the clock of the vertex v_{r+2} takes infinitely the following sequence of values: $r+1, r+2$. In this execution, v_{r+2} executes a rule infinitely often while vertices from p_0 to p_r are never enabled. Note that v_{r+1} is not enabled when $c_{v_{r+2}} = r+2$, hence this vertex is never infinitely enabled. In conclusion, this execution is allowed by the weakly fair daemon. Note that this execution starves v_{r+1}, which contradicts the liveness property of π.

Case 1.3 : v_{r+2} is enabled and the enabled rule modifies its clock into r.
The execution of this rule leads to case 2.

Case 2 : The enabled rule for v_{r+2} modifies its clock into r.
As the daemon is central, v_{r+2} is the only activated vertex and after its execution the new configuration satisfies one of the the following cases:

Case 2.1 : v_{r+2} is not enabled.
If an execution started from γ_2, then no vertex is enabled in any configuration of this execution, which contradicts the liveness property of π.

Case 2.2 : v_{r+2} is enabled and the enabled rule modifies its clock into $r+1$.
Let σ be an execution starting from γ_2 that contains only actions of v_{r+2} (its clock takes infinitely the following value sequence : $r+1, r$). In this execution, v_{r+2} executes a rule infinitely often (by construction) and vertices from v_0 to v_r are never enabled. Note that v_{r+1} is not enabled when $c_{v_{r+2}} = r$, so this vertex is never infinitely enabled. In conclusion, this execution satisfies the weakly fair daemon properties.

Note that this execution starves v_{r+1}, that contradicts the liveness property of π.

Case 2.3 : v_{r+2} is enabled and the enabled rule modifies its clock into $r+2$.
The execution of these rule leads to case 1.

Overall, the only two possible cases (cases 1.3 and 2.3) are the following:

1. v_{r+2} is enabled for modifying its clock value into r when $c_{v_{r+2}} = r+2$ and $c_{v_{r+1}} = r+1$.
2. v_{r+2} is enabled for modifying its clock value into $r+2$ when $c_{v_{r+2}} = r$ and $c_{v_{r+1}} = r+1$.

Let σ be an execution starting from γ_3 that contains only actions of v_{r+2} (its clock takes infinitely the following sequence of values: $r+2, r$). In this execution, v_{r+2} executes a rule infinitely often (by construction) and vertices in $v_0, \ldots, v_r$ are never enabled. Note that v_{r+1} is not enabled when $c_{v_{r+2}} = r+2$, so this vertex is never infinitely enabled. In conclusion, this execution satisfies the weakly fair daemon properties.

This execution starves v_{r+1}, that contradicts the liveness property of π and proves the result. ∎

Theorem 10.3

For any natural number r, there exists no minimal $(1, r)$-ftss distributed protocol for $spec_{AU}$ under the central weakly fair daemon.

Proof: Let r be a natural integer. Assume that there exists a minimal $(1, r)$-ftss distributed protocol π for $spec_{AU}$ under the central weakly fair daemon. By Lemma 10.2, an arbitrary vertex v is not enabled by π if it has only one neighbor v' and if $c_v = c_{v'}$.

Let g be a communication graph reduced to a chain of 2 vertices v and v'. Let γ be a configuration of g where $c_v = c_{v'}$ with no crashed vertex. Notice that no vertex is enabled in γ that contradicts the liveness property of π and proves the result. ∎

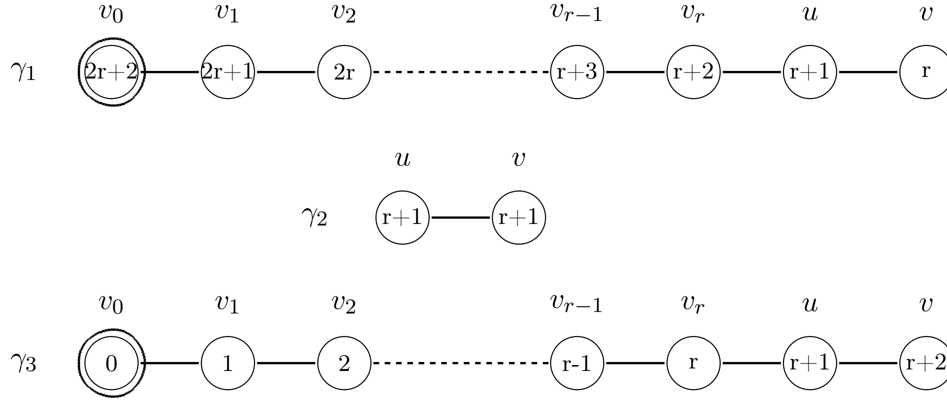

Figure 10.2: The three configurations used in the proof of Lemma 10.3 (the numbers represent clock values and the double circles represent crashed vertices).

10.2.2 Strongly Fair Daemon and Maximal Degree greater than 3

In this section we prove that there exists no minimal $(1, r)$-ftss distributed protocol for $spec_{AU}$ under the central strongly fair daemon if the maximal degree of the communication graph is at least 3.

In order to prove this impossibility result, we use the following property: if a vertex v has only one neighbor u such that $c_u = r + 1$ and if $|c_v - c| \leq 1$, then v is enabled by any minimal $(1, r)$-ftss distributed protocol for $spec_{AU}$ (see Lemma 10.3). Then we construct a strongly fair infinite execution that starves a vertex such that the closest crashed vertex is at more than r hops away. This execution contradicts the liveness property of the distributed protocol (see Theorem 10.4).

Lemma 10.3

Let π be a minimal $(1, r)$-ftss distributed protocol for $spec_{AU}$. If a vertex v has only one neighbor u such that $c_u = r + 1$ and if $|c_v - c_u| \leq 1$, then v is enabled by π.

Proof: Assume that there exists a minimal distributed protocol π that is $(1, r)$-ftss for $spec_{AU}$. Let g be a communication graph that contains at least one vertex v that has only one neighbor u. Assume that $c_u = r+1$ and $|c_v - c_u| \leq 1$. Then, we have:

1. If $c_v = r$, then v is enabled by at least one rule of π. Otherwise, all vertices are starved in the communication graph reduced to the chain $v_0, \ldots, v_r, u, v$ in the configuration γ_1 defined by $\forall i \in \{0, \ldots, r\}, c_{v_i} = 2r + 2 - i$, $c_u = r + 1$, $c_v = r$ where v_0 is crashed (see Figure 10.2) since no correct vertex is enabled (by Lemma 10.1).
2. If $c_v = r + 1$, then v is enabled by at least one rule of π. Otherwise, all vertices are starved in the communication graph reduced to the chain u, v in the configuration γ_2 defined by $c_u = c_v = r+1$ and where no vertex is crashed (see Figure 10.2). Indeed, the symmetry of the configuration implies that u is enabled if and only if v is enabled.

3. If $c_v = r+2$, then v is enabled by at least one rule of π. Otherwise, all vertices are starved in the communication graph reduced to the chain $v_0, \dots, v_r, u, v$ in the configuration γ_3 defined by $\forall i \in \{0, \dots, r\}, c_{v_i} i$, $c_u = r+1$, $c_v = r+2$ and v_0 crashed (see Figure 10.2) since no correct vertex is enabled (by Lemma 10.1). ■

Theorem 10.4

For any natural number r, there exists no minimal $(1,r)$-ftss distributed protocol for $spec_{AU}$ under the central strongly fair daemon if the communication graph has a maximal degree of at least 3.

Proof: Let r be a natural number. Assume that there exists a minimal $(1,r)$-ftss distributed protocol π for $spec_{AU}$ under the central strongly fair daemon for a communication graphs with a degree of at least 3. Let g be the communication graph defined by: $V = \{v_0, \dots, v_{r+1}, u, u'\}$ and $E = \{\{v_i, v_{i+1}\}, i \in \{0, \dots, r\}\} \cup \{\{v_{r+1}, u\}, \{v_{r+1}, u'\}\}$.

As π is deterministic and the communication graph is anonymous, u and u' must behave identically if they have the same clock value (in this case, their local configurations are identical). If $c_{v_{r+1}} = r+1$ and $|c_{v_{r+1}} - c_u| \leq 1$, there exists three local configurations for u: (i) $c_u = r$, (ii) $c_u = r+1$, or (iii) $c_u = r+2$ (the same property holds for u').

By Lemma 10.3, vertex u (respectively u') is enabled in any configuration where $c_{v_{r+1}} = r+1$ and $|c_{v_{r+1}} - c_u| \leq 1$ (respectively $|c_{v_{r+1}} - c_{u'}| \leq 1$). Moreover, in this case, the enabled rule for u (respectively u') modifies its clock into a value in $\{r, r+1, r+2\} \setminus \{c_u\}$ (respectively $\{r, r+1, r+2\} \setminus \{c_{u'}\}$) by the closure property of π.

For each of the three possible local configurations for u or u', π can only allow 2 moves. Hence, there exists 8 possible moves for π. Let us denote each of these possibilities by a triplet (a, b, c) where a, b and c are the clock value of u after the allowed move when $c_u = r$, $c_u = r+1$, and $c_u = r+2$ respectively. Note that, due to the determinism of π, moves allowed for u' and u are identical. There exists the following cases:

Case 1: $(r+1, r, r)$

Let γ_1 be the configuration of g defined by: $\forall i \in \{0, \dots, r+1\}, c_{v_i} = 2r+2-i$, $c_u = r+1$ and $c_{u'} = r$ and v_0 crashed (see Figure 10.3). Note that only u and u' are enabled (by Lemma 10.1). Assume u executes its rule. Hence, its clock takes the value r. By Lemma 10.1, only u and u' are enabled. Assume now that u' executes its rule. Its clock takes the value $r+1$. This configuration is identical to γ_1 (since vertices are anonymous), we can repeat the above reasoning in order to obtain an infinite execution where vertices $v_1, \dots, v_{r+1}$ are never enabled (see Figure 10.4 for an illustration when $r = 1$).

Case 2: $(r+1, r+2, r)$

Let γ_2 be the configuration of g defined by: $\forall i \in \{0, \dots, r+1\}, c_{v_i} i$, $c_u = r$ and $c_{u'} = r+2$ and v_0 crashed (see Figure 10.3). Note that only u and u' are enabled (by Lemma 10.1). Assume u executes its rule. Its clock takes the value $r+1$. By Lemma 10.1, only u and u' are enabled. Assume u executes its rule again. Its clock takes the value $r+2$. By Lemma 10.1, only u and u'

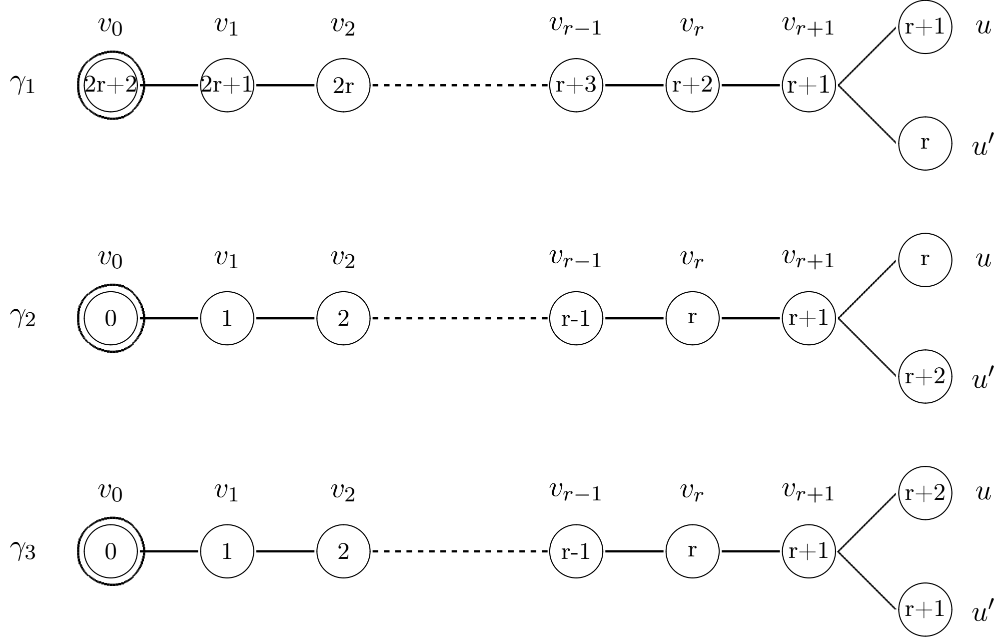

Figure 10.3: The three configurations used in the proof of Theorem 10.4 (the numbers represent clock values and the double circles represent crashed vertices).

are enabled. Assume now that u' executes its rule. Its clock takes the value r. This configuration is identical to γ_2 (since vertices are anonymous). We can repeat the reasoning in order to obtain an infinite execution where vertices in $v_1, \ldots, v_{r+1}$ are never enabled.

Case 3: $(r+1, r, r+1)$
Similar to the reasoning of case 1.

Case 4: $(r+1, r+2, r+1)$
Let γ_3 be the configuration of g defined by: $\forall i \in \{0, \ldots, r+1\}, c_{v_i} = i$, $c_u = r+2$ and $c_{u'} = r+1$ and where v_0 is crashed (see Figure 10.3). Note that only u and u' are enabled (by Lemma 10.1). Assume u' executes its rule. Its clock takes the value $r+2$. By Lemma 10.1, only u and u' are enabled. Assume now that u executes its rule. Its clock takes the value $r+1$. This configuration is identical to γ_3 (since vertices are anonymous). We can repeat the reasoning in order to obtain an infinite execution where vertices in $v_1, \ldots, v_{r+1}$ are never enabled.

Case 5: $(r+2, r, r)$
Let γ_2 be the configuration of g as defined in the case 2 above. Note that only u and u' are enabled (by Lemma 10.1). Assume u executes its rule. Its clock takes the value $r+2$. By Lemma 10.1, only u and u' are enabled. Assume now that u' executes its rule. Its clock takes the value r. This configuration is identical to γ_2 (since vertices are anonymous). We can repeat the reasoning in order to obtain an infinite execution where vertices $v_1, \ldots, v_{r+1}$ are never enabled.

Case 6: $(r+2, r+2, r)$
The reasoning is similar to the case 5.

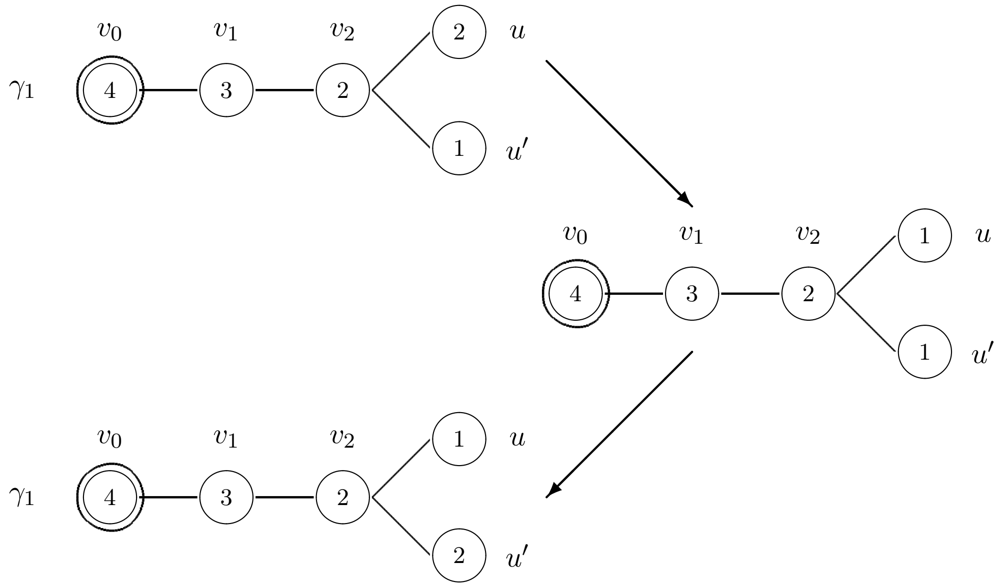

Figure 10.4: Example of the execution constructed in case 1 of Theorem 10.4 when $r = 1$ (the numbers represent clock values and the double circles represent crashed vertices).

Case 7: $(r+2, r, r+1)$
Let γ_2 be the configuration of g as defined in the case 2 above. Note that only u and u' are enabled (by Lemma 10.1). Assume u executes its rule. Its clock takes the value $r+2$. By Lemma 10.1, only u and u' are enabled. Assume u' executes its rule. Its clock takes the value $r+1$. By Lemma 10.1, only u and u' are enabled. Assume u' executes again its rule. Its clock takes the value r. This configuration is identical to γ_2 (since vertices are anonymous). We can repeat the above scenario in order to obtain an infinite execution where vertices $v_1, \ldots, v_{r+1}$ are never enabled.

Case 8: $(r+2, r+2, r+1)$
The proof is similar to the case 4.

Overall, we can construct an infinite execution where vertex v_0 is crashed, vertices from v_1 to v_{r+1} are never enabled and vertices u and u' execute a rule infinitely often. This execution satisfies the strongly fair daemon properties. Notice that in this execution v_{r+1} is never enabled, hence it is starved. This contradicts the liveness property of π and proves the result. ■

10.3 Priority Unison Related Results

In this section, we prove impossibility results similar to those of Section 10.2 whenever the considered unison is priority instead of minimal. In other words, we prove that it is impossible to provide a priority FTSS distributed protocol for $spec_{AU}$ when the daemon is weakly fair (see Section 10.3.1) or when the daemon is strongly

Figure 10.5: Initial configuration used in the proof of Theorem 10.5 (the numbers represent clock values and the double circles represent crashed vertex).

fair and the maximal degree of the communication graph is greater than 3 (see Section 10.3.2).

10.3.1 Weakly Fair Daemon

The main result of this section is that there exists no priority $(1, r)$-ftss distributed protocol for $spec_{AU}$ under the central weakly fair daemon for any natural number r (see Theorem 10.5). We prove this result by contradiction. We construct an execution starting from the configuration γ_0^0 shown in Figure 10.5 allowed by the central weakly fair daemon. We prove that this execution starves p_{r+1} that contradicts the liveness property of the distributed protocol.

Theorem 10.5

For any natural number r, there exists no priority $(1, r)$-ftss distributed protocol for $spec_{AU}$ under the central weakly fair daemon.

Proof: Let r be a natural number. Assume that there exists a priority $(1, r)$-ftss distributed protocol π for $spec_{AU}$ under the central weakly fair daemon. Let g be the communication graph reduced to a chain of length $r+2$. Assume that vertices in g are labeled from left to right as follows: $v_0, v_1, \ldots, v_{r+2}$. Let γ_0^0 be a configuration such that v_0 is crashed and $\forall i \in \{0, \ldots, r+2\}, c_{v_i} = i$ (See Figure 10.5). Note that all the other variables may have any value.

We construct a fragment of execution $\sigma_0' = (\gamma_0^0, \gamma_1^0)(\gamma_1^0, \gamma_2^0) \ldots (\gamma_r^0, \gamma_{r+1}^0)$ starting from γ_0^0 such that $\forall i \in \{0, 1, \ldots, r\}$, the action $(\gamma_i^0, \gamma_{i+1}^0)$ contains only an action of v_{i+1} if v_{i+1} is enabled by π in γ_i^0. By Lemma 10.1, this fragment does not modify the clock value of any vertex in $\{v_0 \ldots v_{r+1}\}$.

We now construct a fragment of execution, σ_0'', starting from γ_{r+1}^0. Let σ_0'' be ε (empty word) if v_{r+2} is not enabled by π in γ_{r+1}^0. Otherwise (v_{r+2} is enabled by π in γ_{r+1}^0), let us define σ_0'' in the following way:

Case 1: There exists a rule of v_{r+2} enabled in γ_{r+1}^0 that does not modify the clock value of v_{r+2}.
Let σ_0'' be $(\gamma_{r+1}^0, \gamma_{r+2}^0)$ where action $(\gamma_{r+1}^0, \gamma_{r+2}^0)$ contains only the execution of this rule by v_{r+2}.

Case 2: Any enabled rule of v_{r+2} in γ_{r+1}^0 modifies its clock value.
Note that the closure property of π implies that the clock of v_{r+2} takes the value r or $r+1$. Let us study the following cases.

Case 2.1: There exists a rule of v_{r+2} enabled in γ_{r+1}^0 that modifies its clock value into $r+1$.
Since π is a priority unison, there exists by definition a fragment of execution $\sigma_0'' = (\gamma_{r+1}^0, \gamma_{r+2}^0) \ldots (\gamma_{r+k-1}^0, \gamma_{r+k}^0)$ that contains only actions of

v_{r+2} such that (i) v_{r+2} executes one of the rules that modifies its clock value into $r+1$ in the action $(\gamma^0_{r+1}, \gamma^0_{r+2})$, (ii) in the actions from γ^0_{r+2} to γ^0_{r+k-1} the clock value of v_{r+2} is not modified, and (iii) in the action $(\gamma^0_{r+k-1}, \gamma^0_{r+k})$ the clock value of v_{r+2} is incremented.

Case 2.2: Any enabled rule of v_{r+2} in γ^0_{r+1} modifies its clock value into r.

Since π is a priority unison, there exists by definition a fragment of execution $\sigma_a = (\gamma^0_{r+1}, \gamma^0_{r+2}) \dots (\gamma^0_{r+k-1}, \gamma^0_{r+k})$ that contains only actions of v_{r+2} such that (i) v_{r+2} executes one of the rules that modifies its clock value into r in the action $(\gamma^0_{r+1}, \gamma^0_{r+2})$, (ii) in the actions from γ^0_{r+2} to γ^0_{r+k-1} the clock value of v_{r+2} is not modified, and (iii) in the action $(\gamma^0_{r+k-1}, \gamma^0_{r+k})$ the clock of v_{r+2} takes the value $r+1$.

Since π is a priority unison, there exists by definition a fragment of execution $\sigma_b = (\gamma^0_{r+k}, \gamma^0_{r+k+1}) \dots (\gamma^0_{r+j-1}, \gamma^0_{r+j})$ that contains only actions of v_{r+2} such that (i) in the actions from γ^0_{r+k} to γ^0_{r+j-1} the clock value of v_{r+2} is not modified and (ii) in the action $(\gamma^0_{r+j-1}, \gamma^0_{r+j})$ the clock value of v_{r+2} is incremented.

Let σ''_0 be $\sigma_a\sigma_b$.

In all cases, we construct a fragment of execution $\sigma_0 = \sigma'_0\sigma''_0$ such that its last configuration (let us denote it by γ^1_0) satisfies: the value of any clock is identical to the one in γ^0_0 (the others variables may have changed). Then, we can reiterate the reasoning and obtain a fragment of execution $\sigma_1, \sigma_2 \dots$ (starting respectively from $\gamma^1_0, \gamma^2_0, \dots$) that satisfies the same property.

We finally obtain an execution $\sigma = \sigma_0\sigma_1 \dots$ that satisfies:

- No vertex is infinitely enabled without executing a rule (since any enabled vertex in γ^i_0 execute a rule or is neutralized during σ_i). Consequently σ is an execution that satisfies the weakly fair daemon properties.
- The clock of vertex v_{r+1} never changes (whereas $dist(g, v_0, v_{r+1}) = r+1$).

This execution contradicts the liveness property of π that is a priority $(1,r)$-ftss distributed protocol for $spec_{AU}$ under the central weakly fair daemon by hypothesis, that proves the result. ■

10.3.2 Strongly Fair Daemon and Maximal Degree greater than 3

The second main result of this section is that there exists no priority $(1,r)$-ftss distributed protocol for $spec_{AU}$ under the central strongly fair daemon for any natural number r if the maximal degree of the communication graph is at least 3. (see Theorem 10.6). We prove this result by contradiction. We construct an execution starting from the configuration γ^0_0 of Figure 10.6 satisfying strongly fair daemon properties that starves v_{r+1}, that contradicts the liveness of the distributed protocol.

Theorem 10.6

For any natural number r, there exists no priority $(1,r)$-ftss distributed protocol for $spec_{AU}$ under the central strongly fair daemon if the communication graph has a maximal degree of at least 3.

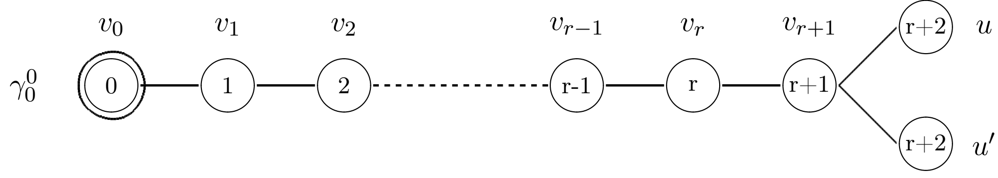

Figure 10.6: The initial configuration for the proof of Theorem 10.6 (the numbers represent clock values and the double circles represent crashed vertices).

Proof: Let r be a natural number. Assume that there exists a priority $(1, r)$-ftss distributed protocol π for $spec_{AU}$ under the central strongly fair daemon even if the communication graph has a maximal degree of at least 3. Let g be the communication graph defined by: $V = \{v_0, \dots, v_{r+1}, u, u'\}$ and $E = \{\{v_i, v_{i+1}\}, i \in \{0, \dots, r\}\} \cup \{\{v_{r+1}, u\}, \{v_{r+1}, u'\}\}$. Note that g has a maximal degree equal to 3.

Let γ_0^0 be the following configuration of g: $\forall i \in \{0, \dots, r+1\}, c_{v_i} = i$, $c_u = c_{u'} = r+2$ and v_0 crashed (see Figure 10.6). Note that, for any execution σ starting from γ_0^0, one of the vertices u and u' must be enabled to modify its clock in a finite time (otherwise the system would be starved following Lemma 10.1). This implies the existence of a fragment of execution $\sigma_a^0 = (\gamma_0^0, \gamma_1^0) \dots (\gamma_{k-1}^0, \gamma_k^0)$ satisfying the following properties:

1. $k \geq 1$ if there exists $i \in \{0, \dots, r+1\}$ such that v_i is enabled in γ_0^0, $k = 0$ otherwise;
2. σ_a^0 contains no modification of clock values; and
3. γ_k^0 is the first configuration where u or u' is enabled to modify its clock value.

Assume now that the scheduling of σ_a^0 satisfies the following property: at each action, the daemon chooses the vertex that has the least recent activation among enabled vertices. Note that this scenario is compatible with the central strongly fair daemon.

Let us study the following cases:

Case 1: u is enabled by π in γ_k^0 for a modification of its clock value. The closure property of π implies that the value of c_u should be modified either to r or to $r+1$.

Case 1.1: The value of c_u is modified to r.

Since π is a priority unison, there exists by definition a fragment of execution $\sigma_{b1}^0 = (\gamma_k^0, \gamma_{k+1}^0) \dots (\gamma_{k+r-1}^0, \gamma_{k+r}^0)$ that contains only actions of u such that (i) in the actions from γ_k^0 to γ_{k+r-1}^0 the clock value of u is not modified and (ii) in the action $(\gamma_{k+r-1}^0, \gamma_{k+r}^0)$ the clock value of u is incremented.

Since π is a priority unison, there exists by definition a fragment of execution $\sigma_{b2}^0 = (\gamma_{k+r}^0, \gamma_{k+r+1}^0) \dots (\gamma_{k+j-1}^0, \gamma_{k+j}^0)$ that contains only executions of a rule by u such that (i) in the actions from γ_{k+r}^0 to γ_{k+j-1}^0 the clock value of u is not modified and (ii) in the action $(\gamma_{k+j-1}^0, \gamma_{k+j}^0)$ the clock value of u is incremented.

Let σ_b^0 be $\sigma_{b1}^0 \sigma_{b2}^0$.

Number of faults	Daemon	Maximal degree	Unison	Impossibility result
$f \geq 2$	Any	Any	Any	Theorem 10.1
$f = 1$	Unfair	Any	Any	Theorem 10.2
	Weakly fair	Any	Minimal	Theorem 10.3
			Priority	Theorem 10.5
	Strongly fair	$deg(g) \geq 3$	Minimal	Theorem 10.4
			Priority	Theorem 10.6

Table 10.1: Summary of impossibility results

Case 1.2: The value of c_u is modified to $r + 1$.
Since π is a priority unison, there exists by definition a fragment of execution $\sigma_b^0 = (\gamma_k^0, \gamma_{k+1}^0) \dots (\gamma_{k+r-1}^0, \gamma_{k+r}^0)$ that contains only actions of u such that (i) in the actions from γ_k^0 to γ_{k+r-1}^0 the clock value of u is not modified and (ii) in the action $(\gamma_{k+r-1}^0, \gamma_{k+r}^0)$ the clock value of u is incremented.

If u' is enabled in the last configuration of σ_b^0 [1], we can construct σ_c^0 similarly to σ_b^0 using vertex u'. Otherwise, let σ_c^0 be ε (the empty word).

Case 2: u' is enabled by π in γ_k^0 for a modification of its clock value.
We can construct σ_b^0 and σ_c^0 similarly as in Case 1 by reversing the roles of u and u'.

Let us define $\sigma^0 = \sigma_a^0 \sigma_b^0 \sigma_c^0$. Notice that the clock values are identical in the first and in the last configuration of σ^0. This implies that we can infinitely repeat the previous reasoning in order to obtain an infinite execution $\sigma = \sigma^0 \sigma^1 \dots$ that satisfies:

- No correct vertex is infinitely often enabled without executing a rule (since u and u' execute a rule infinitely often and others vertices are chosen in function of their least recent execution of a rule, that implies that an infinitely often enabled vertex executes a rule in a finite time). This execution satisfies strongly fair daemon properties.
- The clock value of v_{r+1} is never modified (whereas $dist(g, v_0, v_{r+1}) = r + 1$).

This execution contradicts the liveness property of π, that implies the result. ■

10.4 Summary of Impossibility Results

Table 10.1 summarizes all assumptions of each impossibility result of this chapter. These results show us that it is practically vain to want to solve the asynchronous unison problem in a fault tolerant and self-stabilizing way and *a fortiori* in a strictly stabilizing way.

Nevertheless, we present in the following chapter a minimal and priority strictly stabilizing distributed protocol for asynchronous unison that is optimal with respect to these impossibility results and with respect to stabilization time.

1. In this case, u' was already enabled in the last configuration of σ_a^0

Chapter 11

Strictly Stabilizing Solution

> Don't watch the clock; do what it does. Keep going.
>
> Sam Levenson

Contents

11.1 Strictly Stabilizing Solution **141**
  11.1.1 Distributed Protocol Description 142
  11.1.2 Correctness Proof 143
11.2 Optimality of Convergence Time **149**
  11.2.1 Upper bound 149
  11.2.2 Lower Bound 150
  11.2.3 Conclusion 155

In the previous chapter, we present a broad class of impossibility results regarding FTSS asynchronous unison. These impossibility results imply obviously similar impossibility results about strict stabilization (see Chapter 4).

This chapter focuses on possibility results about strictly stabilizing asynchronous unison. More precisely, we concentrate on classical and simplest solutions to asynchronous unison, namely minimal and/or priority ones (see Section 9.1.3). Then, impossibility results of Chapter 4 imply that we must restrict ourselves at distributed protocol running under strongly fair daemon and on communication graphs reduced to chains or rings.

The contribution of this chapter is twofold. First, we provide a minimal and priority $(1,0)$-strictly stabilizing distributed protocol for $spec_{AU}$ under a strongly fair daemon on communication graphs reduced to chains or rings in Section 11.1. In other words, this distributed protocol is optimal with respect to containment radius and with respect to impossibility results of Chapter 4. Then, we prove in Section 11.2 that the stabilization time to this distributed protocol is optimal for its class.

11.1 Strictly Stabilizing Solution

In this section, we present the minimal and priority $(1,0)$-strictly stabilizing distributed protocol for $spec_{AU}$ in Section 11.1.1. This distributed protocol is called $\mathcal{SSU}$ for $\mathcal{S}$trictly $\mathcal{S}$tabilizing $\mathcal{U}$nison. Recall that $\mathcal{SSU}$ is designed for a strongly fair daemon and a communication graph reduced to a chain or to a ring with at most one

Byzantine vertex. In order to complete the proof of the strict stabilization of this distributed protocol, we must assume that the daemon is locally central. Note that this assumption is not proved necessary. Then, Section 11.1.2 proves the correctness of our distributed protocol.

11.1.1 Distributed Protocol Description

The distributed protocol can operate on communication graphs reduced to either chain or ring. For the description of the distributed protocol, let us introduce some topological terminology. A middle vertex has two neighbors. An end vertex has only one. In a ring, every vertex is a middle vertex. A chain has two end vertices. We consider the communication graph to be laid out horizontally left to right. We, therefore, speak of left and right neighbors for a vertex and left and right ends of a chain. This global orientation of the chain is only assumed for the purposes of exposition, we do not require that the local orientation of vertices is globally consistent (that is, the labeling of right and left neighbor is arbitrary for each vertex of the communication graph).

Recall that clock drift between two correct neighbors is the absolute value of the difference between their clock values. Two neighboring vertices u and v are in unison if the drift between them is no more than 1. An island is a segment of correct vertices such that for each vertex u, if its neighbor u is also in this island, then v and u are in unison. A vertex with no in-unison neighbors is assumed to be a single-vertex island. Note that a Byzantine vertex never belongs to an island. The width of an island is the number of vertices in this island.

The main idea of the distributed protocol is as follows. Vertices form islands (of vertices with synchronized clocks by definition). The distributed protocol is designed such that the clocks of the vertices with adjacent islands drift closer to each other and the islands eventually merge. If a Byzantine vertex restricts the drift of one such island, for example by never changing its clock, the other islands still drift and synchronize with the affected island.

Operation description A description of $\mathcal{SSU}$ is shown in Protocol 11.1. Specifically, $\mathcal{SSU}$ operates as follows. Each vertex v maintains a single variable c_v where it stores its current clock value. That is, our distributed protocol is minimal.

We grouped the vertex rules into end vertex rules and middle vertex rules. Middle vertex rules are further grouped into: operation (executed when the vertex is in unison with at least one of its neighbors) and synchronization (executed otherwise).

At least one rule is always enabled at an end vertex. Depending on the clock value of its neighbor, the left end vertex either increments or decrements its own clock using rules *endLeftUp* and *endLeftDown*. The operation of the right end vertex is similar.

Let us describe the rules of a middle vertex. If vertex v is in unison with its left neighbor, v can adjust c_v to match its right neighbor using rules *middleLeftUp* or *middleLeftDown*. The execution of neither rule breaks the unison of v and its left

Protocol 11.1 $\mathcal{SSU}$: Minimal and priority $(1,0)$-strictly stabilizing distributed protocol for $spec_{AU}$ on chains and rings for vertex v.

Constants

l, r: left and right neighbors of v (this labeling is arbitrary and we do not require that it is consistent with the one of neighborhood vertices)
$deg(g, v)$: degree of v

Variable

c_v: natural number, clock value of v

Rules

/* End vertex rules */
endLeftUp :: $(deg(g,v) = 1) \wedge ((c_v = c_r) \vee (c_v = c_r - 1)) \longrightarrow c_v := c_v + 1$
endLeftDown :: $(deg(g,v) = 1) \wedge ((c_v \geq c_r + 1) \vee (c_v < c_r - 1)) \longrightarrow c_v := c_r - 1$
endRightUp and *endRightDown* are similar
/* Middle vertex operation rules */
middleLeftUp :: $(deg(g,v) = 2) \wedge (c_v = c_l \vee c_v = c_l - 1) \wedge (c_v \leq c_r) \longrightarrow c_v := c_v + 1$
middleLeftDown :: $(deg(g,v) = 2) \wedge (c_v = c_l \vee c_v = c_l + 1) \wedge (c_v > c_r) \longrightarrow c_v := c_v - 1$
middleRightUp and *middleRightDown* are similar
/* Middle vertex synchronization rules */
syncUp :: $(deg(g,v) = 2) \wedge (c_v < c_l - 1) \wedge (c_v < c_r - 1) \longrightarrow c_v := min\{c_l, c_r\}$
syncDown :: $(deg(g,v) = 2) \wedge (c_v > c_l + 1) \wedge (c_v > c_r + 1) \longrightarrow c_v := max\{c_l, c_r\}$

neighbor. Similar adjustment is done for the left neighbor using *middleRightUp* and *middleRightDown*. Note that if v is in unison with both of its neighbors and c_l and c_r differ by 2, none of these rules of v are enabled as any changes of c_v break the unison with a neighbor of v (this property is shared by any minimal $(1,0)$-strictly stabilizing distributed protocol for $spec_{AU}$, see Lemma 10.1). If v is in unison with neither of its neighbors, and the clocks of the two neighbors are either both greater or both less than the clock of v, the vertex synchronizes its clock with one of the neighbors using rule *syncDown* or *syncUp*.

Note that the construction of rules *endLeftUp*, *endRightUp*, *middleRightUp*, and *middleLeftUp* ensures the priority of $\mathcal{SSU}$. Indeed, if there exists a vertex v such that for any $u \in N_v$, $(c_u = c_v$ or $c_u = c_v + 1)$, then one of these rules is enabled for v. If the daemon chooses to activate v, we obtain a portion of execution satisfying Definition 9.3 since each of these rules increment c_v by one.

Example operation The operation of our distributed protocol is best understood with an example. Figures 11.1 and 11.2 illustrates the operation of $\mathcal{SSU}$ on a chain respectively without and with a Byzantine vertex. Figures 11.3 and 11.4 show the operation of $\mathcal{SSU}$ on rings respectively without and with a Byzantine vertex.

11.1.2 Correctness Proof

We provide the proof of the strict stabilization of $\mathcal{SSU}$ in this section. Recall that we consider only executions allowed by the locally central strongly fair daemon. We split this proof in two parts. The first one focuses on chains while the second one is interested in rings but main ideas are similar in both cases. The proof relies on the

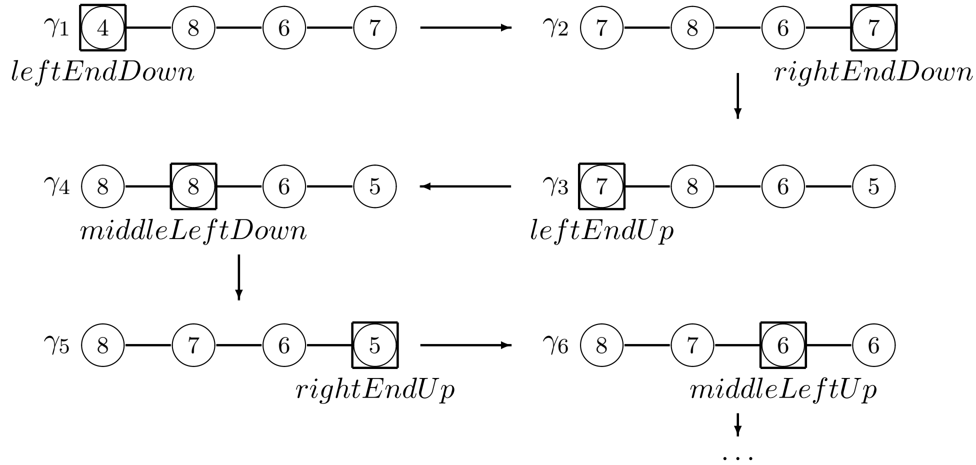

Figure 11.1: An example operation sequence of $\mathcal{SSU}$ on a chain with no faults. Numbers represent clock values. Squared vertex has an enabled rule to be executed.

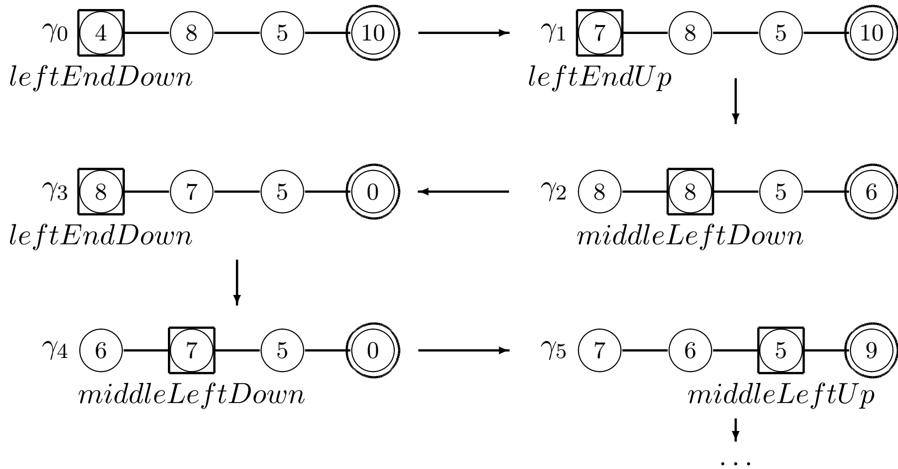

Figure 11.2: An example operation sequence of $\mathcal{SSU}$ on a chain with a Byzantine vertex. Numbers are vertex clock values. The Byzantine vertex is in double circle. Squared vertex has an enabled rule to be executed.

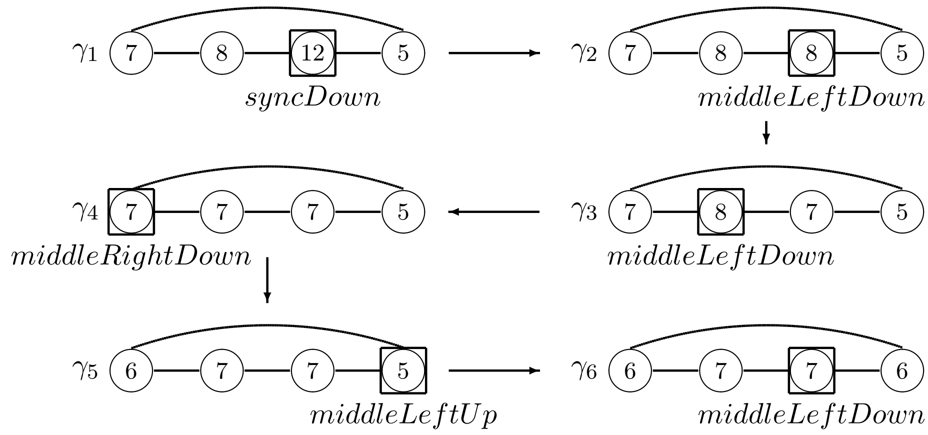

Figure 11.3: An example operation sequence of $\mathcal{SSU}$ on a ring with no faults. Numbers represent clock values. Squared vertex has an enabled rule to be executed.

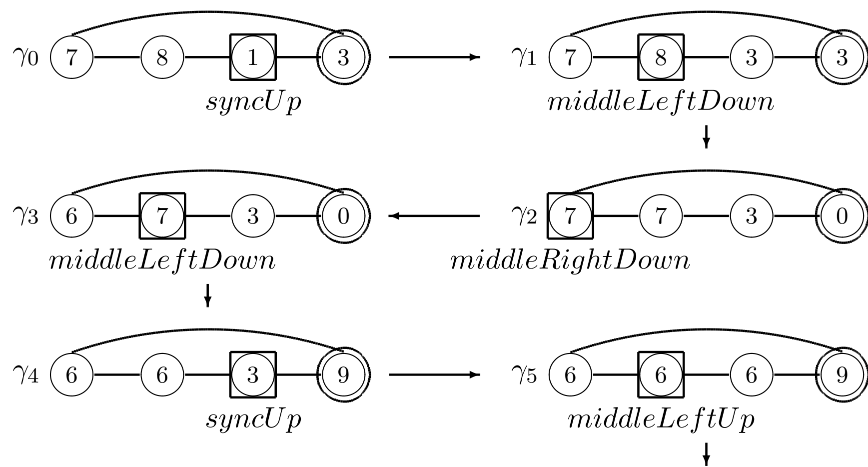

Figure 11.4: An example operation sequence of $\mathcal{SSU}$ on a chain with a Byzantine vertex. Numbers are vertex clock values. The Byzantine vertex is in double circle. Squared vertex has an enabled rule to be executed.

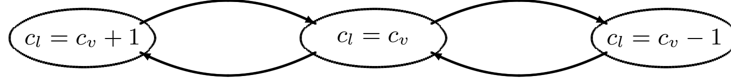

Figure 11.5: The transitions of in-unison neighbor vertices l and v. An illustration for the proof of Lemma 11.2.

following facts. First, once two vertices are in the same island, they remain so during the whole execution due to the construction of the distributed protocol. Then, we can prove that there always exists an island in which each vertex is infinitely often activated (in other words, the progress is guaranteed for at least one island). We can use this property to prove that two adjacent islands eventually merged. Finally, an induction proof on the initial number of islands shows us the convergence of $\mathcal{SSU}$.

Chains For chains it is sufficient to consider executions of the distributed protocol for the case where the Byzantine vertex is at the end of the chain. Indeed, if the Byzantine vertex is in the middle of the chain, the synchronization of the two segments of correct vertices is independent of each other due to the problem specification. Thus, without loss of generality, we assume that if there exists a Byzantine vertex in the system, it is the right end vertex.

Lemma 11.1

If an execution of $\mathcal{SSU}$ on a chain starts from a configuration where two vertices v and u belong to the same island, then the two vertices belong to the same island in every configuration of this execution.

Lemma 11.1 states that an island is never broken. The validity of the lemma can be easily ascertained by the examination of the distributed protocol's rules as a correct vertex never de-synchronizes from its in-unison neighbors.

Lemma 11.2

In every execution of $\mathcal{SSU}$ on a chain, each vertex in the leftmost island executes a rule infinitely often.

Proof: The proof is by induction on the width of the leftmost island. In every configuration, the left end vertex has either *endLeftUp* or *endLeftDown* enabled. Due to the strongly fair daemon, this vertex executes a rule infinitely often in any execution.

Assume that the left neighbor l of a vertex v that belongs to the leftmost island executes a rule infinitely often in an execution σ. According to Lemma 11.1, l and v are in unison in every configuration of σ. That is, l and v transition between the three sets of states: $c_l = c_v + 1$, $c_l = c_v$ and $c_l = c_v - 1$. See Figure 11.5 for illustration. Observe that, regardless of the clock value of the right neighbor of v, if $c_l = c_v$ then v has either *middleLeftUp* or *middleLeftDown* rule enabled. If v executes this rule, the system goes either in the state where $c_l = c_v + 1$ or $c_l = c_v - 1$. Since l executes infinitely often a rule in σ a configuration where $c_l = c_v$ repeats infinitely often. That is, one of v's rules is enabled infinitely often in σ. Since the daemon is strongly fair, v executes a rule infinitely often. ■

Lemma 11.3

If an execution of $\mathcal{SSU}$ on a chain starts from a configuration where a vertex v belongs to the leftmost island while its right correct neighbor r does not, then this execution contains a configuration where both v and r belong to the same island.

So, Lemma 11.3 claims that every two adjacent islands eventually merge.

Proof: We prove the lemma by demonstrating that the drift between v and r decreases to one in every execution of $\mathcal{SSU}$. Let us consider the rules of r. The execution of any rule by r can only decrease the drift between the two vertices. The execution of the rules by v always decreases the drift as well. According to Lemma 11.2, v executes infinitely often a rule in any execution. This means that any execution contains a configuration where the drift between v and r is zero. ■

Define the following predicate for any configuration γ:

$$INV(\gamma) \equiv \textit{ each correct vertex is in unison with its correct neighbors in } \gamma$$

Lemma 11.4

The predicate INV is closed by rules of $\mathcal{SSU}$ and any execution of $\mathcal{SSU}$ on chains reaches a configuration satisfying INV in a finite time.

Proof: If a configuration satisfy *INV*, then every correct vertex is in unison with its correct neighbors by definition, all correct vertices belong to a single island. The closure of *INV* follows from Lemma 11.1.

Note that Lemma 11.3 guarantees that the two leftmost islands eventually merge. The convergence of any execution of $\mathcal{SSU}$ to a configuration satisfying *INV* can be proven by induction on the number of islands in the initial configuration. ■

Proposition 11.1

$\mathcal{SSU}$ is a minimal and priority $(1,0)$-strictly stabilizing distributed protocol for $spec_{AU}$ under the locally central strongly fair daemon on any communication graph reduced to a chain.

Proof: Lemma 11.4 allows us to state that any execution of $\mathcal{SSU}$ on a chain reaches in a finite time a configuration satisfying *INV*. Then, the safety property of $spec_{AU}$ follows immediately from the closure of *INV* proved in Lemma 11.4. Let us consider the liveness property. Once in unison the only rule that a vertex can execute on its clock is increment or decrement. According to Lemma 11.2, every correct vertex is infinitely often activated. Since the clock values are natural numbers, each vertex is bound to execute an infinite number clock increments. Hence the liveness. ■

Rings Since there are no end vertices on a ring, we only have to consider the middle vertex rules. The proof on rings shares similarities with the one on chains that allows us to present it more quickly.

Lemma 11.5

If an execution of $\mathcal{SSU}$ on a ring starts from a configuration where two vertices v and u belong to the same island, then the two vertices belong to the same island in every configuration of this execution.

The above lemma is proven similarly to Lemma 11.1.

Lemma 11.6

In every execution of $\mathcal{SSU}$ on a ring, there is an island where every vertex is infinitely often activated.

Proof: Observe that in every configuration of $\mathcal{SSU}$ on a ring, there are always a largest and a smallest clock value. Hence, there is at least one correct vertex whose clock holds the largest or the smallest value in the system. Indeed, in the worst case, the Byzantine vertex holds only one of them. This correct vertex has a rule enabled. Consequently, in every configuration of $\mathcal{SSU}$ on a ring, there exists at least one enabled correct vertex and then, there are infinitely many rules executed by correct vertices in every execution of $\mathcal{SSU}$ since we consider a strongly fair daemon. Since there are finitely many correct vertices, at least one correct vertex is infinitely often activated. Let us consider the island to which this vertex belongs. The rest of the lemma is proven by induction on the width of this island similar to Lemma 11.2. ■

Lemma 11.7

If an execution of $\mathcal{SSU}$ starts from a configuration where there is more than one island, then there exists two neighboring vertices v and u that belong to two distinct islands in this configuration such that this execution contains a configuration where both v and u belong to the same island.

Proof: Let us consider the initial configuration of $\mathcal{SSU}$ on a ring with more than one island. According to Lemma 11.6, there is at least one island in this configuration where every vertex executes a rule infinitely often. Assume, without loss of generality, that this island has an adjacent island to the right. An argument similar to the one employed in the proof of Lemma 11.3 demonstrates that these islands eventually merge. ■

The two results below are similar to their equivalents for chains.

Lemma 11.8

The predicate INV is closed by rules of $\mathcal{SSU}$ and any execution of $\mathcal{SSU}$ on rings reaches a configuration satisfying INV in a finite time.

Proposition 11.2

$\mathcal{SSU}$ is a minimal and priority $(1,0)$-strictly stabilizing distributed protocol for $spec_{AU}$ under the locally central strongly fair daemon on any communication graph reduced to a ring.

Conclusion Proposition 11.1 and 11.2 allows us to state the following result:

Theorem 11.1

$\mathcal{SSU}$ is a minimal and priority $(1,0)$-strictly stabilizing distributed protocol for $spec_{AU}$ under the locally central strongly fair daemon on any communication graph reduced to a chain or a ring.

Note that features of the distributed protocol $\mathcal{SSU}$ claimed by this theorem are optimal with respect to containment radius and with respect to impossibility results of Chapter 10 at the notable exception of the distribution of the daemon. Indeed, our distributed protocol needs a locally central daemon whereas no impossibility result in Chapter 10 proves the necessity of this assumption. The question to perform a similar distributed protocol under a distributed daemon is still open.

In the following section, we study stabilization time of $\mathcal{SSU}$ and prove that it is optimal.

11.2 Optimality of Convergence Time

In this section, we compute the stabilization time of $\mathcal{SSU}$. We estimate the stabilization time in the number of asynchronous rounds [DIM97b, BDPV07]. In general, this notion is somewhat tricky to define for strongly fair daemon, at the actions of vertices may become disabled and then enabled an arbitrary many times before execution. However, this definition simplifies for the case of $\mathcal{SSU}$ as every correct vertex executes a rule infinitely often (by Lemmas 11.2 and 11.6). We define an asynchronous round to be the smallest portion of an execution of the distributed protocol where every correct vertex executes (at least) a rule.

11.2.1 Upper bound

First, we show that $\mathcal{SSU}$ needs at most L rounds to stabilize where L is the largest clock drift between two correct neighbors in the initial configuration of the system. This result follows from the fact that the two neighboring vertices that has the largest initial clock drift can be initially in unison with their other neighbors. In this way, they can reduce the clock drift from at most one at each activation. Hence, we can deduce the upper bound on the stabilization time of $\mathcal{SSU}$.

Proposition 11.3

The stabilization time of $\mathcal{SSU}$ is in $O(L)$ asynchronous rounds both on chains and rings where L is the maximum clock drift between two correct neighbors in the initial configuration.

Proof: Assume that there exists an execution σ such that there exists at least two distinct islands I_1 and I_2 at the end of the round L_σ (where L_σ is the maximum clock drift between two correct neighbors in the initial configuration of σ). Note that $L_\sigma \geq 2$. Otherwise, any vertex is in unison with its neighbor in the initial configuration and Lemma 11.1 or 11.5 implies I_1 and I_2 are never distinct.

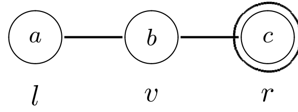

Figure 11.6: Configuration used in proof of Lemma 11.9 (clock values appear inside vertices and the double circles represent Byzantine vertex).

Let u and v be two neighbor vertices such that $u \in I_1$ and $v \in I_2$. Without loss of generality, we can assume that $c_v < c_u$ in the initial configuration of σ. By construction, we have $c_u - c_v \leq L_\sigma$.

While I_1 and I_2 are distinct, according to the proof of Lemma 11.3 or 11.7, the following property holds: $c_v < c_u$.

In the case where the communication graph is a chain, note that u and v are not end vertices. Otherwise, u and v are in unison at the end of the first round since the end vertex synchronizes its clock with the one of its neighbor at its first activation and this contradicts the construction of σ and the fact that $L_\sigma \geq 2$.

Now, we can observe that any activation of u by a middle vertex operation or synchronization rule can only decrease the clock value of u by at least one.

Following the definition of asynchronous round, there is at least one activation of u during each round of σ. Then, we can conclude that, at the end of the round i ($1 \leq i \leq L_\sigma$), we have: $c_u - c_v \leq L_\sigma - i$.

We can deduce that u and v are necessarily in unison at the end of the round $L_\sigma - 1$ that contradicts the construction of σ. Then, the stabilization time of $\mathcal{SSU}$ is in $O(L)$ asynchronous rounds both on chains and rings. Hence the result. ■

11.2.2 Lower Bound

In this section, we show that any deterministic minimal $(1, 0)$-strictly stabilizing distributed protocol for $spec_{AU}$ on a chain or on a ring under the central strongly fair daemon needs at least L rounds to stabilize where L is the largest clock drift between two correct neighbors in the initial configuration of the system.

Indeed, we prove that any such distributed protocol shares some properties with $\mathcal{SSU}$. In particular, two neighboring vertices in unison must maintain this unison and a vertex in unison with only one of its neighbors moves its clock closer to the one of its second neighbor in a finite time. These two properties allow us to prove that these distributed protocols have (at least) the same convergence time than $\mathcal{SSU}$.

Lower bound on chains In the following lemmas, π denotes any deterministic minimal $(1, 0)$-strictly stabilizing distributed protocol for $spec_{AU}$ on the chain under a central strongly fair daemon. As previously, results are provided for a central daemon for the sake of generality.

Lemma 11.9

When a middle vertex is in unison with only one of its neighbors, any enabled rule of π for this vertex maintains this unison.

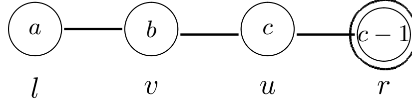

Figure 11.7: Configuration used in proof of Lemma 11.10 (clock values appear inside vertices and the double circles represent Byzantine vertex).

Proof: Assume that there exists a set of clock values $\{a, b, c\}$ (with $|a - b| \leq 1$ and $|b - c| \geq 2$) such that a middle vertex v is enabled by a rule R of π when $c_v = b$ and neighbors clock are respectively a and c and that R modifies c_v into a value b' (with $|a - b'| \geq 2$).

Then, consider the following initial configuration: $V = \{l, v, r\}$, $E = \{\{l, v\}, \{v, r\}\}$, r is a Byzantine vertex and $c_l = a$, $c_v = b$, $c_r = c$ (see Figure 11.6). We can observe that this configuration satisfies INV. By construction, v is enabled by R in this configuration (recall that π is minimal and deterministic). If the daemon chooses v, then we obtain a configuration which does not satisfy INV. Hence, π does not respect the closure of $spec_{AU}$. This is contradictory with its construction. ■

Lemma 11.10

When a middle vertex v is in unison with only one of its neighbors (denote by u the other neighbor of v), the following property holds: in any execution starting from this configuration in which u remains not synchronized with v, v moves its clock closer to the clock of u in a finite time.

Proof: Assume that there exists a set of clock values $\{a, b, c\}$ (with $|a - b| \leq 1$ and $|b - c| \geq 2$) such that there exists an execution σ of π starting from a configuration (in which $c_v = b$ and neighbors clock values are respectively a and c – denote by u the vertex such that $c_u = c$) in which u remains not synchronized with v and in which v never moves its clock closer to the clock of u.

We deal with the case where $b > c$ (the case where $b < c$ is similar). Then, consider the following initial configuration γ: $V = \{l, v, u, r\}$, $E = \{\{l, v\}, \{v, u\}, \{u, r\}\}$, r is a Byzantine vertex and $c_l = a$, $c_v = b$, $c_u = c$, $c_r = c - 1$ (see Figure 11.7). If r acts as a crashed vertex, its clock value remains constant. Then, by Lemma 11.9, we have $c_u \in \{c, c-1, c-2\}$ in any configuration of any execution starting from γ. Hence, u cannot distinguish this execution from σ (recall that π is minimal and deterministic). Consequently, there exists an execution starting from γ such that $c_v \geq b$ and $c_u \leq c$ in any state. This contradicts the convergence property of π. ■

Lemma 11.11

When an end vertex is in unison with its neighbor, there exists an enabled rule of π for this vertex.

Proof: Assume that there exists a set of clock values $\{a, b\}$ (with $|a - b| \leq 1$) such that an end vertex v is not enabled by any rule of π when $c_p = a$ and its neighbor clock is b.

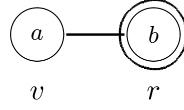

Figure 11.8: Configuration used in proof of Lemma 11.11 (clock values appear inside vertices and the double circles represent Byzantine vertex).

Then, consider the following initial configuration: $V = \{v, r\}$, $E = \{\{v, r\}\}$, r is a Byzantine vertex and $c_v = a$, $c_r = b$ (see Figure 11.8). By construction, v is not enabled by π in this configuration (recall that π is minimal and deterministic). Assume now that r acts as a crashed vertex. Then, we can observe that v is never enabled by π in this execution, that contradicts the liveness property of π. ■

If we consider the execution described in the proof of Lemma 11.11, we can observe that v is infinitely often activated (by fairness assumption) and that its clock is always in the set $\{b-1, b, b+1\}$ (by closure of π). Since π is minimal and deterministic, we can deduce that values of c_v over this execution follow a given cycle. We characterize now π by this cycle. More formally, we say that:

1. π is of type **1** if the cycle is $b, b+1, b, b+1, \ldots$.
2. π is of type **2** if the cycle is $b, b-1, b, b-1, \ldots$.
3. π is of type **3** if the cycle is $b, b+1, b-1, b, b+1, b-1, \ldots$.

Notice that the distributed protocol $\mathcal{SSU}$ is of type **1** using this characterization.

Proposition 11.4

The stabilization time of any deterministic minimal $(1,0)$-strictly stabilizing distributed protocol for $spec_{AU}$ on chains under the central strongly fair daemon is in $\Omega(L)$ asynchronous rounds where L is the maximum clock drift between two correct neighbors in the initial configuration.

Proof: Assume that π is a deterministic minimal $(1,0)$-strictly stabilizing distributed protocol for $spec_{AU}$ on chains under the central strongly fair daemon . We provide the proof of this proposition in the case where π is of type **1** (other cases are similar).

Let a and t be two natural numbers. Consider the following initial configuration γ_0: $V = \{v, u, r, b\}$, $E = \{\{v, u\}, \{u, r\}, \{r, b\}\}$, b is a Byzantine vertex and $c_v = a+2t$, $c_u = a+2t$, $c_r = a$, $c_b = a$ (see Figure 11.9). Hence, we have a maximal clock drift between two correct neighbors of $L = 2t$.

Note that v is enabled by π to take the value $a+2t+1$ in γ_0 (by Lemma 11.11 and the fact that π is minimal and of type **1**). By Lemmas 11.10, 11.9, and the fact that π is minimal, we can deduce that u is enabled by π to take the value $a+2t-1$ only when $c_p = a+2t$. Similar reasoning holds for r which is enabled to take the value $a+1$ when $c_b = a$.

Then, the following portion execution of π starting from γ_0 is possible: v is activated and its clock takes value $a+2t+1$, v is activated and its clock takes value $a+2t$ (v is enabled by Lemma 11.11 and the new value is determined by the type of π), u is activated and its clock takes value $a+2t-1$, r is activated and its clock

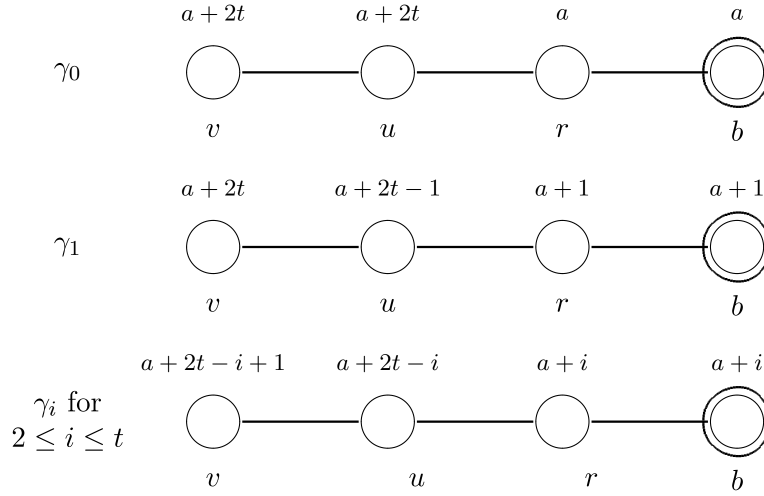

Figure 11.9: Configurations used in proof of Proposition 11.4 (clock values appear over vertices and the double circles represent Byzantine vertex).

takes value $a+1$ and finally, d takes the value $a+1$ (recall that b is a Byzantine vertex). We obtain the configuration γ_1 depicted in Figure 11.9.

We can observe that the first round R_1 of our execution ends in γ_1 and that we have now a maximal clock drift between two correct neighbors of $a+2(t-1)$.

By the same reasoning, we can construct a sequence of $t-1$ rounds $R_2, \dots, R_t$ where R_i begin in γ_{i-1} and ends in γ_i ($2 \leq i \leq t$) as follows: v is activated and its clock takes value $a+2t+1-i$, u is activated and its clock takes value $a+2t-i$, r is activated and its clock takes value $a+i$ and finally, b takes the value i. We obtain the configuration γ_i at the end of round R_i ($2 \leq i \leq t$) depicted in Figure 11.9. At the end of round R_i ($2 \leq i \leq t$), we have a maximal clock drift between two correct neighbors of $2(t-i)$.

We can conclude that, at the end of the round R_{t-1}, the maximal clock drift between two correct neighbors is 2 whereas, at the end of the round R_t, the maximal clock drift between two correct neighbors is 1 (since we have $c_v - c_u = 1$ and $c_u - c_r = 0$). By construction of t, we can conclude that π needs $\Omega(L)$ asynchronous rounds to stabilize. ■

Lower bound on rings. In the following lemmas, π denotes any deterministic minimal $(1,0)$-strictly stabilizing distributed protocol for $spec_{AU}$ on a ring under the central strongly fair daemon.

Lemma 11.12

When a vertex is in unison with only one of its neighbors, any enabled rule of π for this vertex maintains this unison.

Proof: The proof of Lemma 11.9 directly applies here if we consider the following communication graph: $V = \{v, u, r\}$ and $E = \{\{v,u\}, \{u,r\}, \{r,v\}\}$. ■

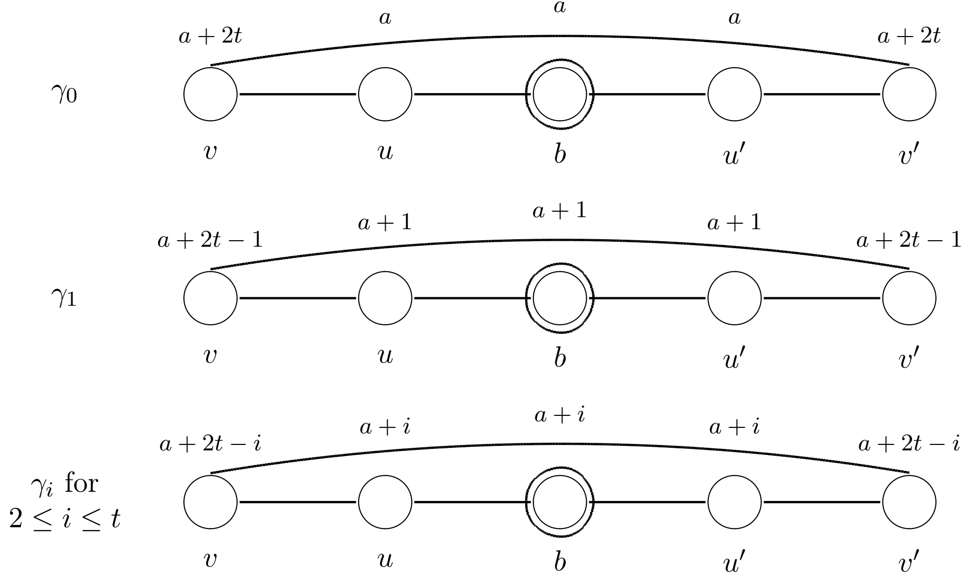

Figure 11.10: Configurations used in proof of Proposition 11.5 (clock values appear over vertices and the double circles represent Byzantine vertex).

Lemma 11.13

When a vertex v is in unison with only one of its neighbors (denote by u the other neighbor of v), the following property holds: in any execution starting from this configuration in which u remains not synchronized with v, v moves its clock closer to the clock of u in a finite time.

Proof: The proof of Lemma 11.10 directly applies here if we consider the following system: $V = \{v, u, r, b\}$ and $E = \{\{v, u\}, \{u, r\}, \{r, b\}, \{b, u\}\}$. ■

Proposition 11.5

The stabilization time of any deterministic minimal $(1, 0)$-strictly stabilizing distributed protocol for $spec_{AU}$ on rings under the central strongly fair daemon is in $\Omega(L)$ asynchronous rounds where L is the maximum clock drift between two correct neighbors in the initial configuration.

Proof: Assume that π is a deterministic minimal $(1, 0)$-strictly stabilizing distributed protocol for $spec_{AU}$ on rings under the central strongly fair daemon .

Let a and t be two natural numbers. Consider the following initial configuration γ_0: $V = \{v, u, b, u', v'\}$, $E = \{\{v, u\}, \{u, b\}, \{b, u'\}, \{u', v'\}, \{v', v\}\}$, b is a Byzantine vertex and $c_v = c_{v'} = a + 2t$, $c_u = c_{u'} = c_b = a$ (see Figure 11.10). Hence, we have a maximal clock drift between two correct neighbors of $L = 2t$.

Note that v and v' are enabled by π to take the value $a + 2t - 1$ in γ_0 (by Lemmas 11.13 and 11.12 and the fact that π is minimal). By similar reasoning, we can deduce that u and u' are enabled to take the value $a + 1$.

Then, the following portion of execution of π starting from γ_0 is possible: v is activated and its clock takes value $a+2t-1$, v' is activated and its clock takes value $a+2t-1$, u is activated and its clock takes value $a+1$, u' is activated and its clock takes value $a+1$ and finally, the clock of b takes the value $a+1$ (recall that b is a Byzantine vertex). We obtain the configuration γ_1 depicted in Figure 11.10.

We can observe that the first round R_1 of our execution ends in γ_1 and that we have now a maximal clock drift between two correct neighbors of $a+2(t-1)$.

By the same reasoning, we can construct a sequence of $t-1$ rounds $R_2,\dots,R_t$ where R_i starts in γ_{i-1} and ends in γ_i $(2 \leq i \leq t)$ as follows: v is activated and its clock takes value $a+2t-i$, v' is activated and its clock takes value $a+2t-i$, u is activated and its clock takes value $a+i$, u' is activated and its clock takes value $a+i$ and finally, the clock of b takes the value $a+i$ (recall that b is a Byzantine vertex). We obtain the configuration γ_i at the end of round R_i $(2 \leq i \leq t)$ depicted in Figure 11.10. At the end of round R_i $(2 \leq i \leq t)$, we have a maximal clock drift between two correct neighbors of $2(t-i)$.

We can conclude that, at the end of the round R_{t-1}, the maximal clock drift between two correct neighbors is 2 whereas, at the end of the round R_t, the maximal clock drift between two correct neighbors is 0. By construction of t, we can conclude that π needs $\Omega(L)$ asynchronous rounds to stabilize. ■

11.2.3 Conclusion

Let us review our conclusions so far. Proposition 11.3 proves that the stabilization complexity of $\mathcal{SSU}$ is in $O(L)$ asynchronous rounds while Propositions 11.4 and 11.5 show that any $(1,0)$-strictly stabilizing distributed protocol requires at least that many asynchronous rounds to stabilize. The following theorem summarizes these results.

Theorem 11.2

The stabilization complexity of $\mathcal{SSU}$ is optimal. It stabilizes in $\Theta(L)$ asynchronous rounds where L is the largest clock drift between two correct neighbors in the initial configuration.

Chapter 12

Conclusion of Part III

I must govern the clock, not be governed by it.

Golda Meir

Contents

12.1 Summary of Contributions **157**
12.2 Concluding Remarks **158**

12.1 Summary of Contributions

In the third part of this thesis, we studied asynchronous unison protocols subject to arbitrary transient and intermittent Byzantine fault patterns. Intuitively, asynchronous unison consists in a weak synchronization of digital clocks. We must ensure that any vertex has a clock difference of at most one with any of its neighbors in a finite time and that, from this point, any correct vertex clock is infinitely often incremented while it maintaining its synchronization invariant with its neighbors.

First, we proved in Chapter 10 some impossibility results related to FTSS asynchronous unison. By observations made in Chapter 4, these impossibility results imply similar impossibility results for the case of strictly stabilizing asynchronous unison. More precisely, we proved impossibility results related to the number of crashed vertices, to fairness of daemon, to some properties of the distributed protocol (namely, minimality and priority), and/or to some classes of communication graphs. Table 12.1 summarizes all impossibility results of Chapter 10. These results show that most of the interesting cases (from a practical point of view) are impossible to solve.

<table>
<tr><th>Number of faults</th><th>Daemon</th><th>Maximal degree</th><th>Unison</th><th>Impossibility result</th></tr>
<tr><td>$f \geq 2$</td><td>Any</td><td>Any</td><td>Any</td><td>Theorem 10.1</td></tr>
<tr><td rowspan="5">$f = 1$</td><td>Unfair</td><td>Any</td><td>Any</td><td>Theorem 10.2</td></tr>
<tr><td rowspan="2">Weakly fair</td><td rowspan="2">Any</td><td>Minimal</td><td>Theorem 10.3</td></tr>
<tr><td>Priority</td><td>Theorem 10.5</td></tr>
<tr><td rowspan="2">Strongly fair</td><td rowspan="2">$deg(g) \geq 3$</td><td>Minimal</td><td>Theorem 10.4</td></tr>
<tr><td>Priority</td><td>Theorem 10.6</td></tr>
</table>

Table 12.1: Summary of impossibility results of Chapter 10

Nevertheless, we explored joint tolerance to transient and to intermittent Byzantine faults for the asynchronous unison problem in Chapter 11. We presented a minimal and priority $(1, 0)$-strictly stabilizing distributed protocol for this problem on chain and ring communication graphs under the locally central strongly fair daemon. From impossibility results of Chapter 10, we can observe that all our assumptions for this distributed protocol are necessary except for possible weakening of the distribution of the daemon. Moreover, we prove that our distributed protocol has an optimal stabilization time in $\Theta(L)$ asynchronous rounds where L is the maximal clock drift between two correct neighbors in the initial configuration.

12.2 Concluding Remarks

To conclude this part, we propose first a generalization of our problem and show that our results still apply to this new class of problems. Then, we present some questions that are still open.

Generalization: κ-asynchronous unison In this paragraph, we briefly explain how to generalize the above results to a weaker problem. Assume that $\kappa \in \mathbb{N}^*$. In the κ-asynchronous unison problem (denote its specification by $spec_{\kappa-AU}$), a drift of at most κ units is allowed between clocks of any two correct neighbors. Hence, $spec_{AU}$ corresponds to $spec_{1-AU}$.

Let us observe that a similar result to Lemma 10.1 holds for $spec_{\kappa-AU}$:

Lemma 12.1

Let π be a (f, r)-ftss distributed protocol for $spec_{\kappa-AU}$ (under any central daemon). Let γ be a configuration where a vertex v with $c_v \geq \kappa$ has two neighbors u and u' such that: $c_u = c_v - \kappa$ and $c_{u'} = c_v + \kappa$. If v executes an action of π during an action (γ, γ'), then this action does not modify the value of c_v. If π is also minimal, then the vertex v is not enabled by π in γ.

As Lemma 10.1 is the basis of impossibility proofs of Chapter 10, we can deduce that all impossibility results presented in Chapter 10 still hold for $spec_{\kappa-AU}$.

In a similar way, it is possible to borrow fundamentals ideas of the distributed protocol $\mathcal{SSU}$ to cope with $spec_{\kappa-AU}$ instead of $spec_{AU}$. Indeed, it is sufficient to define that two correct neighbors are in-unison if their clock drift is smaller than κ and to write a distributed protocol that ensures similar properties as $\mathcal{SSU}$.

Open questions. An immediate future work is to generalize the possibility result (that assumes a locally central daemon) to cope with a distributed daemon, or extend the impossibility proof in that case. There also remains the open case of distributed protocols that neither satisfy the minimality or the priority properties. We conjecture that at least one of those properties is necessary for the purpose of deterministic self-stabilization, yet none of those may be required for deterministic weak stabilization [Gou01] (weak stabilization is a weaker property than self-stabilization

since only existence of execution reaching a legitimate configuration is guaranteed). As recent results [DTY08] hint that weak-stabilizing solutions can be easily turned into probabilistic self-stabilizing ones, this raises the open question of the possibility of probabilistic strict stabilization for dynamic tasks in asynchronous systems.

The existence of a solution for the state model opens another path of research. It is interesting to consider the existence of a solution in lower atomicity models such as shared register or message-passing models (see Section 2.3). We conjecture that a solution in such models is more difficult to obtain as the lower atomicity tends to empower Byzantine vertices. Indeed, in the shared-register model a Byzantine vertex may report differing clock values to its right and left neighbor. Such behavior makes a single fault ring communication graph essentially equivalent to two faults in the chain communication graph. The latter is proven unsolvable. Hence, we conjecture that in the lower atomicity models, the only communication graph that allows a solution to asynchronous unison is the chain.

Part IV

Spanning Tree

Chapter 13

Introduction of Part IV

Trees sprout up just about everywhere in computer science...

Donald E. Knuth

Contents

13.1 Problem and Related Works **164**
13.1.1 Related Works 164
13.1.2 Specification 166
13.2 Contributions of Part IV **170**
13.3 Containing Byzantine Faults in Self-Stabilization **172**
13.3.1 Strict Stabilization 172
13.3.2 Strong Stabilization 173
13.3.3 Topology-Aware Stabilization 175
13.3.4 Discussion 177

In the fourth part of this thesis, we interest in a fundamental building block of distributed systems: spanning tree construction. As stated in Section 2.1, distributed systems are in part characterized by the fact that vertices must communicate together to solve almost any task (at least interesting ones). Therefore, the optimization of communications is at the core of many distributed protocols. There exists several way to optimize communications in a distributed system. A simple way is to minimize the number of communication links used by vertices to communicate with each other. In this way, the spanning tree is the best communication structure since it ensures by definition the possibility of communication between any pair of vertices while it minimizes the number of communication links. This optimality of the spanning tree explains that there exists numerous distributed protocols for constructing spanning trees as building blocks for more complex tasks that need some communication optimization.

For a given communication graph, there exists a lot of different spanning trees (for example a complete communication graph admits n^{n-2} different spanning trees). Even if any of these spanning tree minimizes by definition the number of communication links, all are not equivalent. Indeed, we can consider some other criteria to distinguish spanning trees. For example, a breadth-first search (BFS) spanning tree [HC92, DIM93, AKY90] allows us to minimize the delay of communication between any vertex and a distinguished one (called the root of the tree) and a minimum

weight spanning tree [GHS83] minimizes the global cost of the tree (in the case of weighted communication graph).

In this part, we focus on a large class of spanning tree constructions: the maximum metric spanning tree construction with respect to any maximizable metric [GS03]. Intuitively, a metric is a scheme to compute a distance along any path of the communication graph. A metric is maximizable if there always exists a spanning tree that maximizes the metric of each vertex of any communication graph with respect to a distinguished vertex called the root. For example, the shortest path [TH94] or the flow metric [GS94] are maximizable. In contrast, there exists no maximizable metric to model the minimum weight [GHS83] or the minimum degree [BB04, BPBR11] spanning tree construction. The large span of this class of metrics motivates some previous works [GS99, GS03].

This chapter aims to introduce this part by first presenting related works and the formal specification of the problem (see Section 13.1). Then, Section 13.2 summarizes the contributions of the Part IV. We finally present some definitions used in the sequel of this part in Section 13.3.

13.1 Problem and Related Works

This section motivates our work about maximum metric spanning tree construction in distributed systems subject to any transient and intermittent Byzantine fault pattern by providing a short survey about spanning tree construction in self-stabilizing area (see Section 13.1.1). Then, we present the formal definition of a maximal metric spanning tree and we specify our problem in Section 13.1.2.

13.1.1 Related Works

As spanning tree construction has been extensively studied, it is quite impossible to do an exhaustive survey on this subject. In the following, we present fundamental results only.

Note that spanning tree construction was first studied in centralized systems. This problem is an important part of graph theory. There exists centralized protocols as well for simple properties of spanning trees (*e.g.* depth-first search spanning tree, breadth-first search spanning tree [Cor01], or shortest path spanning tree [Dij71]) than for more complicated ones (*e.g.* minimum weight spanning tree [Kru56, Pri57] or minimum degree spanning tree [FR92]).

In the same way, spanning tree construction was extensively studied in the context of distributed systems either in a fault-free setting or in presence of faults. In fault-free distributed systems, there exists number of adaptations of centralized protocols to construct spanning trees with respect to numerous properties (*e.g.* minimum weight spanning tree [GHS83], minimum degree spanning tree [BB04], or Steiner trees [CHK93]).

Self-stabilizing protocols Gärtner proposes in [Gär03] a good survey on self-stabilizing distributed protocols for spanning tree construction for the three most simple properties: depth-first search spanning tree, breadth-first spanning tree and shortest path spanning tree.

The first self-stabilizing distributed protocol to construct a depth-first search spanning tree was by Collin and Dolev [CD94]. Note that any self-stabilizing token circulation (see *e.g.* [HW97, DJPV00]) may be used to construct such a spanning tree. Regarding breadth-first search spanning tree construction, the first self-stabilizing solution was by Dolev, Israeli and Moran [DIM90, DIM93] but a simpler one was provided by [HC92] (we provide a full description of this latter in Section 14.2). Finally, self-stabilizing solutions to shortest path spanning tree construction may be found in [HL02, BK07] for example. We refer the interested reader to [Rov09] for a detailed comparison of these self-stabilizing distributed protocols for spanning tree construction.

Note that there also exists self-stabilizing distributed for more complex properties of spanning trees as the minimum diameter spanning tree [BLB95], minimum degree spanning tree [BPBR11], or Steiner spanning tree [BPBR09].

Finally, Ghosh, Gupta, Herman and Pemmaraju study fault-containing spanning tree construction in [GGHP96]. This fault containment means that the stabilization time of the distributed protocol depends on the number of corrupted vertices in the initial configuration, not that this distributed protocol tolerates to permanent faults (see Section 4.1.2).

Maximizable metrics In [GS03], Gouda and Schneider define a large class of spanning tree constructions using the concept of maximizable metric. In this work, the metric of a path of the communication graph is the result of the application of the metric operator to the value to each edge of the path. For example, the shortest path metric associates a weight (a natural number) to each edge of the communication graph and the metric of a path is computed by the sum of the weight of each edge of the path. A metric is maximizable if there always exists a spanning tree that maximizes the metric of each vertex of any communication graph with respect to a distinguished vertex called the root.

This concept of maximizable metric enclosed a lot of classical metrics as breadth-first search, shortest path or flow metrics (defined in [Sch97]), that justifies the interest in maximum metric spanning tree construction. The main results of [GS03] is a full characterization of maximizable metrics that we formally provided in Section 13.1.2.

A self-stabilizing distributed protocol for maximum metric spanning tree construction with respect to any maximizable metric is provided by [GS99]. In this distributed protocol, any vertex try to maximize its metric in the tree by choosing as its parent the neighbor that provide the best metric value. Using this strategy, the arbitrary initial configuration may lead to the formation of cycles of vertices that has the same incorrect metric value. The key idea of this distributed protocol

is to use a hop counter (upper bounded by a given constant D) to detect and break cycles of vertices that have the same (incorrect) maximum metric. The choice of the constant D is obviously capital for the self-stabilization of the distributed protocol. Gouda and Schneider proved that their distributed protocol is self-stabilizing if D is an upper bound on the length of the longest path of the desired spanning tree.

The idea to provide a generic distributed protocol able to compute a spanning tree according to a large class of metric motivates other works. For instance, another formalism that encompasses a different set of metrics is presented in [DT01, DT03, DDT06].

Byzantine tolerance At our knowledge, there exists no distributed protocol for spanning tree construction in presence of both transient and intermittent Byzantine faults even for simple properties as depth-first search spanning tree or shortest path spanning tree.

We propose to fill this gap in this part by studying the maximum metric spanning tree construction that gathers a large class of spanning tree construction as highlighted by [GS03].

13.1.2 Specification

In this section, we formally define maximum (routing) metric trees using formalism introduced by [GS03]. Informally, the goal of a routing distributed protocol is to construct a tree that simultaneously maximizes the metric value of all of the vertices with respect to some total ordering $\prec$. Then, we can specify the problem considered in this part of the thesis.

Maximum metric tree First, we recall definitions and notations of [GS03] and state the main result about characterization of maximizable metric (that is, metrics such that there always exists a tree maximizing the metric of each vertex).

Definition 13.1 (*Routing metric*)

A routing metric (or just metric) $\mathcal{M}$ is a five-tuple $\mathcal{M} = (M, W, met, mr, \prec)$ where:

1. *M is a set of metric values,*
2. *W is a set of edge weights,*
3. *met is a metric function whose domain is $M \times W$ and whose range is M,*
4. *mr is the maximum metric value in M with respect to $\prec$ and is assigned to the root of the system,*
5. *$\prec$ is a less-than total order relation over M that satisfies the following three conditions for arbitrary metric values m, m', and m'' in M:*
   - (a) *irreflexivity: $m \not\prec m$,*
   - (b) *transitivity : if $m \prec m'$ and $m' \prec m''$ then $m \prec m''$,*
   - (c) *totality: $m \prec m'$ or $m' \prec m$ or $m = m'$.*

Any metric value $m \in M \setminus \{mr\}$ *satisfies the utility condition (that is, there exist* $w_0, \ldots, w_{k-1}$ *in* W *and* $m_0 = mr, m_1, \ldots, m_{k-1}, m_k = m$ *in* M *such that* $\forall i \in \{1, \ldots, k\}, m_i = met(m_{i-1}, w_{i-1})$*).*

For instance, we provide below the definition of three classical metrics with this model:

- the shortest path metric ($\mathcal{SP}$) in which the distance from any vertex to the root is minimized. Edge weights are natural numbers and the metric operator is the sum.
- the flow metric ($\mathcal{F}$) in which each vertex chooses the path of maximal flow (*i.e.* the weight of the edge of minimum weight of the path) to the root. Edge weights are natural numbers and the metric operator is the minimum function.
- the reliability metric ($\mathcal{R}$) in which each vertex chooses the path of maximal reliability (*i.e.* the product of edge weights of the path) to the root. Edge weights are real numbers (between 0 and 1) and the metric operator is the product.

Note also that we can model the construction of a spanning tree with no particular constraints in this model using the metric $\mathcal{NC}$ described below and the construction of a BFS spanning tree using the shortest path metric ($\mathcal{SP}$) with $W_1 = \{1\}$ (we denote this metric by $\mathcal{BFS}$ in the following).

$$\begin{array}{rcl}
\mathcal{SP} & = & (M_1, W_1, met_1, mr_1, \prec_1) \\
where & & M_1 = \mathbb{N} \\
& & W_1 = \mathbb{N} \\
& & met_1(m, w) = m + w \\
& & mr_1 = 0 \\
& & \prec_1 \text{ is the classical } > \text{ relation}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{F} & = & (M_2, W_2, met_2, mr_2, \prec_2) \\
where & & mr_2 \in \mathbb{N} \\
& & M_2 = \{0, \ldots, mr_2\} \\
& & W_2 = \{0, \ldots, mr_2\} \\
& & met_2(m, w) = min\{m, w\} \\
& & \prec_2 \text{ is the classical } < \text{ relation}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{R} & = & (M_3, W_3, met_3, mr_3, \prec_3) \\
where & & M_3 = [0, 1] \\
& & W_3 = [0, 1] \\
& & met_3(m, w) = m * w \\
& & mr_3 = 1 \\
& & \prec_3 \text{ is the classical } < \text{ relation}
\end{array}$$

$$
\begin{array}{rcl}
\mathcal{NC} & = & (M_4, W_4, met_4, mr_4, \prec_4) \\
\textit{where} & & M_4 = \{0\} \\
 & & W_4 = \{0\} \\
 & & met_4(m, w) = 0 \\
 & & mr_4 = 0 \\
 & & \prec_4 \ \textit{is the classical} \ < \ \textit{relation}
\end{array}
$$

Definition 13.2 (***Assigned metric***)

An assigned metric over a communication graph g is a six-tuple $(M, W, met, mr, \prec, wf)$ where $(M, W, met, mr, \prec)$ is a metric and wf is a function that assigns to each edge of g a weight in W.

Let a rooted path (from v) be an elementary path from a vertex v to the root r. The next set of definitions are with respect to an assigned metric $(M, W, met, mr, \prec, wf)$ over a given communication graph g.

Definition 13.3 (***Metric of a rooted path***)

The metric of a rooted path in g is the prefix sum of met over the edge weights in the path and mr.

For example, if a rooted path p in g is $v_k, \ldots, v_0$ with $v_0 = r$, then the metric of p is $m_k = met(m_{k-1}, wf(\{v_k, v_{k-1}\}))$ with $\forall i \in \{1, \ldots, k-1\}, m_i = met(m_{i-1}, wf(\{v_i, v_{i-1}\})$ and $m_0 = mr$.

Definition 13.4 (***Maximum metric path***)

A rooted path p from v in g is called a maximum metric path with respect to an assigned metric if and only if for every other rooted path q from v in g, the metric of p is greater than or equal to the metric of q with respect to the total order $\prec$.

Definition 13.5 (***Maximum metric of a vertex***)

The maximum metric of a vertex $v \neq r$ (or simply metric value of v) in g is defined by the metric of a maximum metric path from v. The maximum metric of r is mr.

Definition 13.6 (***Maximum metric spanning tree***)

A spanning tree t of g is a maximum metric spanning tree with respect to an assigned metric over g if and only if every rooted path in t is a maximum metric path in g with respect to the assigned metric.

The goal of the work of [GS03] is the study of metrics that always allow the construction of a maximum metric spanning tree. The definition follows.

Definition 13.7 (***Maximizable metric***)

A metric is maximizable if and only if for any assignment of this metric over any communication graph g, there is a maximum metric spanning tree for g with

respect to the assigned metric.

An interesting result about maximizable metrics due to [GS03] provides a fully characterization of maximizable metrics as follow. First, they define two classes of metrics. A metric is bounded if and only if the application of the metric function to any metric value does not increase it (for any edge weight) whereas a metric is monotonic if and only if the metric function preserves the order $\prec$ on metric values. Formal definitions follow.

Definition 13.8 (*Boundedness*)

A metric $(M, W, met, mr, \prec)$ is bounded if and only if:

$$\forall m \in M, \forall w \in W, met(m, w) \prec m \text{ or } met(m, w) = m$$

Definition 13.9 (*Monotonicity*)

A metric $(M, W, met, mr, \prec)$ is monotonic if and only if:

$$\forall (m, m') \in M^2, \forall w \in W, m \prec m' \Rightarrow (met(m, w) \prec met(m', w) \text{ or } met(m, w) = met(m', w))$$

Then, [GS03] proves that a metric is maximizable if and only if it belongs to the intersection of these two classes of metrics.

Theorem 13.1 (*Characterization of maximizable metrics [GS03]*)

A metric is maximizable if and only if this metric is bounded and monotonic.

Specification Given a maximizable metric $\mathcal{M} = (M, W, mr, met, \prec)$, the aim of this part is to study the construction of a maximum metric spanning tree with respect to $\mathcal{M}$ rooted to a pre-defined vertex r (called the root) in presence of any transient and intermittent Byzantine fault pattern. Note that we must assume that the root vertex is never a Byzantine one. It is obvious that these Byzantine vertices may disturb some correct vertices. Therefore, we relax the problem in the following way: we want to construct a maximum metric spanning forest with respect to $\mathcal{M}$. The root of any tree of this spanning forest must be either the real root or a Byzantine vertex.

Each vertex v has three O-variables: a pointer to its parent in its tree ($prnt_v \in N_v \cup \{\bot\}$), a variable that stores its current metric value ($level_v \in M$) and an integer which stores a distance ($dist_v \in \mathbb{N}$). We use the following specification of the problem.

We introduce new notations as follows. Given an assigned metric $(M, W, met, mr, \prec, wf)$ over the communication graph g and two vertices u and v, we denote by $max_met(g, u, v)$ the maximum metric of vertex u when v plays the role of the root

of the communication graph (that is, when $level_v = mr$). If u and v are neighbors, we denote by $w_{u,v}$ the weight of the edge $\{u, v\}$ (that is, the value of $wf(\{u, v\})$).

Definition 13.10 (*$\mathcal{M}$-path*)

Given an assigned metric $\mathcal{M} = (M, W, mr, met, \prec, wf)$ over a communication graph g, a path $(v_0, \ldots, v_k)$ $(k \geq 1)$ of g is a $\mathcal{M}$-path if and only if:

1. $prnt_{v_0} = \bot$, $level_{v_0} = mr$, $dist_{v_0} = 0$, *and* $v_0 \in B \cup \{r\}$;
2. $\forall i \in \{1, \ldots, k\}, prnt_{v_i} = v_{i-1}$, $level_{v_i} = met(level_{v_{i-1}}, w_{v_i,v_{i-1}})$, *and* $dist_{v_i} = i$;
3. $\forall i \in \{1, \ldots, k\}, met(level_{v_{i-1}}, w_{v_i,v_{i-1}}) = \max_{\prec, u \in N_v}\{met(level_u, w_{v_i,u}), \}$; *and*
4. $level_{v_k} = max_met(g, v_k, v_0)$.

As explained below in Section 13.3, we choose in this part to specify problems in a specific way. We state the specification by a specification predicate for each vertex that is true if this vertex complies to a solution of the problem (note that this specification predicate is not necessarily local). We can now specify the problem of the maximum metric spanning tree construction.

Specification 13.1 (*Maximum metric spanning tree construction $spec_{MMT}$*)

The specification predicate $spec_{MMT}(v)$ of the maximum metric tree construction with respect to a maximizable metric $\mathcal{M}$ for vertex v follows:

$$spec_{MMT}(v) : \begin{cases} prnt_v = \bot \text{ and } level_v = mr, \text{ and } dist_v = 0 \text{ if } v \text{ is the root } r \\ \text{there exists a } \mathcal{M}\text{-path } (v_0, \ldots, v_k) \text{ such that } v_k = v \text{ otherwise} \end{cases}$$

13.2 Contributions of Part IV

Position of Part IV As explained in Section 13.1.1, the spanning tree construction problem received a great attention from the self-stabilizing area. But, at our knowledge, there exists no study of spanning tree construction in distributed systems simultaneously subject to transient and intermittent Byzantine faults (whatever is the considered metric). The goal of this part is to remedy at this fact by providing the first work in this context. We choose to restrict this study to maximum metric spanning tree constructions.

From now, we assume to work in a deterministic distributed system subject to any transient and intermittent Byzantine fault pattern. The communication graph is arbitrary and anonymous but we assume the existence of a distinguished vertex r called the root. Finally, note that all results presented in Part IV assume the state computational model (see Section 2.3.2).

Overview of Part IV Contributions of this part are twofold. First, motivating by impossibility results of strict stabilization for global tasks (as the spanning tree construction) presented in Section 4.2.3, we propose three new concepts for Byzantine containment in self-stabilization. These new concepts, respectively called strong stabilization, topology-aware strict stabilization, and topology-aware strong stabilization, weaken the constraints on the containment radius of strict stabilization in order to by-pass such impossibility results. In strong stabilization, the containment radius is weaken in time since we allow correct vertices outside the containment radius to be disturbed a finite number of times by Byzantine ones after the convergence to a legitimate configuration. In topology-aware stabilization, the weakening is in space since we generalize the containment radius to a containment area. This containment area is simply the set of correct vertices (that is function of the communication graph) that may be infinitely often disturbed by Byzantine vertices. Note that this weakening of containment radius in containment area may be applied both to strict and strong stabilization. Formal definitions of these concepts are presented in Section 13.3.

Then, the sequel of the Part IV is devoted to the proof of the effectiveness of our new concepts of Byzantine containment in self-stabilization using maximum metric spanning tree construction as a benchmark.

1. In Chapter 14, we prove by two case studies that all maximizable metrics are not equivalent with respect to Byzantine containment in self-stabilization. Indeed, we prove that there exists a strongly-stabilizing solution for spanning tree construction (without constraints) while it is impossible to provide a similar distributed protocol for BFS spanning tree construction. We complete these results with a topology-aware stabilizing solution to the BFS spanning tree construction.

2. Chapter 15 generalizes results from the previous one since we provide a full characterization of the set of maximizable metrics that allows strong stabilization. The second contribution of this chapter is the design of a distributed protocol that performs the optimal containment areas for topology-aware strict and strong stabilization for maximum metric spanning construction for any maximizable metric.

Results of Chapter 14 appear in a publication in IEEE Transactions on Parallel and Distributed Systems [DMT11b], in proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2010) [DMT10c], of the 12èmes Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications (Algotel 2010) [DMT10a], and of the 13èmes Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications (Algotel 2011) [DMT11a]. On the other hand, results of Chapter 15 are published in the proceedings of the 24th International Symposium on Distributed Computing (DISC 2010) [DMT10b] and of the 25th International Symposium on Distributed Computing (DISC 2011) [DMT11c].

13.3 Containing Byzantine Faults in Self-Stabilization

According to definitions presented in Section 4.2.3, it is obvious that the maximum metric spanning tree construction admits a $diam(g)$-restrictive specification. Hence, we can deduce, using [NA02] that this problem does not admit any strictly stabilizing solution. This section aims to present some weaker Byzantine containment properties in self-stabilization. The remainder of this part is devoted to the proof of the usefulness of these new containment schemes.

Specification Problems considered in this part are so-called static problems, *i.e.* they require the system to find static solutions. For example, the spanning-tree construction problem is a static problem, while the asynchronous unison problem is not. Some static problems can be defined by a specification predicate (shortly, specification), $spec(v)$, for each vertex v: a configuration is a desired one (with a solution) if every vertex satisfies $spec(v)$. A specification $spec(v)$ is a boolean expression on variables of V_v ($\subseteq V$) where V_v is the set of vertices whose variables appear in $spec(v)$ (note that V_v is not related to N_v, in other words the specification predicate is not necessarily local). The variables appearing in the specification are called output variables (shortly, O-variables).

13.3.1 Strict Stabilization

First, we state the definition of strict stabilization (already introduced in Section 4.2.3) using specification predicate instead of global specification on executions. Note that these two definitions are equivalent but we introduce the new one in order to be consistent with definitions of the remainder of this section.

Following definitions are with respect to a specification predicate *spec*. Given an integer c, a c-correct vertex is a vertex defined as follows.

Definition 13.11 (*c-correct vertex*)

*A vertex is c-correct if it is correct (*i.e. *not Byzantine) and located at distance more than c from any Byzantine vertex.*

Definition 13.12 (*(c, f)-containment*)

A configuration γ is (c, f)-contained for specification spec if, given at most f Byzantine vertices, in any execution starting from γ, every c-correct vertex v always satisfies $spec(v)$ and never changes its O-variables.

The parameter c of Definition 13.12 refers to the containment radius defined in [NA02]. The parameter f refers explicitly to the number of Byzantine processes, while [NA02] dealt with unbounded number of Byzantine faults (that is $f \in \{0 \dots n\}$).

Definition 13.13 (*Strict stabilization*)

A distributed protocol π is (c, f)-strictly stabilizing for specification spec if,

starting from any arbitrary configuration, every execution involving at most f Byzantine vertices contains a configuration that is (c, f)-contained for spec.

13.3.2 Strong Stabilization

To circumvent impossibility results of strict stabilization, we define a weaker notion. Here, the requirement to the containment radius is relaxed, *i.e.* there may exist vertices outside the containment radius that invalidate the specification predicate, due to Byzantine actions. However, the impact of Byzantine triggered action is limited in times: the set of Byzantine vertices may only impact vertices outside the containment radius a bounded number of times, even if Byzantine vertices execute an infinite number of actions.

In the following of this section, we present the formal definition of strong stabilization for a specification predicate *spec*. From the states of c-correct vertices (see Definition 13.11), c-legitimate configurations and c-stable configurations are defined as follows.

Definition 13.14 (*c-legitimate configuration*)

A configuration γ is c-legitimate for spec *if every c-correct vertex v satisfies* spec(v).

Definition 13.15 (*c-stable configuration*)

A configuration γ is c-stable if every c-correct vertex never changes the values of its O-variables as long as Byzantine vertices make no action.

Roughly speaking, the aim of self-stabilization is to guarantee that a distributed protocol eventually reaches a c-legitimate and c-stable configuration. However, a self-stabilizing system can be disturbed by Byzantine vertices after reaching a c-legitimate and c-stable configuration. The c-disruption represents the period where c-correct vertices are disturbed by Byzantine vertices and is defined as follows

Definition 13.16 (*c-disruption*)

A portion of execution $\delta = (\gamma_0, \gamma_1) \ldots (\gamma_{t-1}, \gamma_t)$ $(t > 1)$ is a c-disruption if and only if the following holds:

1. *δ is finite;*
2. *δ contains at least one action of a c-correct vertex for changing the value of an O-variable;*
3. *γ_0 is c-legitimate for* spec *and c-stable; and*
4. *γ_t is the first configuration after γ_0 such that γ_t is c-legitimate for* spec *and c-stable.*

Now we can define a self-stabilizing distributed protocol such that Byzantine vertices may only impact vertices outside the containment radius a bounded number of times, even if Byzantine vertices execute an infinite number of actions.

Definition 13.17 (*(t, k, c, f)-time contained configuration*)

A configuration γ_0 is (t, k, c, f)-time contained for spec if given at most f Byzantine vertices, the following properties are satisfied:

1. *γ_0 is c-legitimate for spec and c-stable;*
2. *every execution starting from γ_0 contains a c-legitimate configuration for spec after which the values of all the O-variables of c-correct vertices remain unchanged (even when Byzantine vertices make actions repeatedly and forever),*
3. *every execution starting from γ_0 contains at most t c-disruptions, and*
4. *every execution starting from γ_0 contains at most k actions of changing the values of O-variables for each c-correct vertex.*

Definition 13.18 (*Strong stabilization*)

A distributed protocol π is (t, c, f)-strongly stabilizing for spec if and only if starting from any arbitrary configuration, every execution involving at most f Byzantine vertices contains a (t, k, c, f)-time contained configuration for spec that is reached after at most ℓ rounds. Parameters ℓ and k are respectively the (t, c, f)-stabilization time and the (t, c, f)-vertex disruption times of π.

Note that a (t, k, c, f)-time contained configuration is a (c, f)-contained configuration when $t = k = 0$, and thus, (t, k, c, f)-time contained configuration is a generalization (relaxation) of a (c, f)-contained configuration. Thus, a strongly stabilizing distributed protocol is weaker than a strictly stabilizing one (as vertices outside the containment radius may take incorrect actions due to Byzantine influence). However, a strongly stabilizing distributed protocol is stronger than a classical self-stabilizing one (that may never meet their specification in the presence of Byzantine vertices).

The parameters t, k and c are introduced to quantify the strength of fault containment, we do not require each vertex to know the values of the parameters. There exists some relationship between these parameters as the following proposition states:

Proposition 13.1

If a configuration is (t, k, c, f)-time contained for spec, then $t \leq nk$.

Proof: Let γ_0 be a (t, k, c, f)-time contained configuration for *spec*. Assume that $t > nk$.

If there exists no execution $\sigma = (\gamma_0, \gamma_1)\ldots$ such that σ contains at least $nk + 1$ c-disruptions, then γ_0 is in fact a (nk, k, c, f)-time contained configuration for *spec* (and hence, we have $t \leq nk$). This is contradictory. So, there exists an execution $\sigma = (\gamma_0, \gamma_1)\ldots$ such that σ contains at least $nk + 1$ c-disruptions.

As any c-disruption contains at least one action of a c-correct vertex for changing the value of an O-variable by definition, we obtain that σ contains at least $nk + 1$ actions of c-correct vertices for changing the values of O-variables. There is at most

n c-correct vertices. So, there exists at least one c-correct vertex which takes at least $k+1$ actions for changing the value of O-variables in σ. This is contradictory with the fact that γ_0 is a (t,k,c,f)-time contained configuration for *spec*. ∎

Discussion between strong stabilization and pseudo stabilization There exists an analogy between the respective powers of (c,f)-strict stabilization and (t,c,f)-strong stabilization for the one hand, and self-stabilization and pseudo-stabilization for the other hand.

A pseudo-stabilizing distributed protocol (defined in [BGM93], see Section 4.1) guarantees that every execution has a suffix that matches the specification, but it could never reach a legitimate configuration from which any possible execution matches the specification. In other words, a pseudo-stabilizing distributed protocol can continue to behave satisfying the specification, but with having possibility of invalidating the specification in future. A particular daemon can prevent a pseudo-stabilizing distributed protocol from reaching a legitimate configuration for arbitrarily long time, but cannot prevent it from executing its desired behavior (that is, a behavior satisfying the specification) for arbitrarily long time. Thus, a pseudo-stabilizing distributed protocol is useful since desired behavior is eventually reached.

Similarly, every execution of a (t,c,f)-strongly stabilizing distributed protocol has a suffix such that every c-correct vertex executes its desired behavior. But, for a (t,c,f)-strongly stabilizing distributed protocol, there may exist executions such that the system never reach a configuration after which Byzantine vertices never have the ability to disturb the c-correct vertices: all the c-correct vertices can continue to execute their desired behavior, but with having possibility that the distributed system (respectively each vertex) could be disturbed at most t (respectively k) times by Byzantine vertices in future. A notable but subtle difference is that the invalidation of the specification is caused only by the effect of Byzantine vertices in a (t,c,f)-strongly stabilizing distributed protocol, while the invalidation can be caused by the daemon in a pseudo-stabilizing distributed protocol.

13.3.3 Topology-Aware Stabilization

We describe here another weaker notion than the strict stabilization: the topology-aware strict stabilization (denoted by TA strict stabilization for short). Here, the requirement to the containment radius is relaxed, *i.e.* the set of vertices which may be disturbed by Byzantine ones is not reduced to the union of c-neighborhood of Byzantine vertices (*i.e.* the set of vertices at distance at most c from a Byzantine vertex) but can be defined depending on the communication graph and Byzantine vertices location.

In the following, we give formal definition of this new kind of Byzantine containment. From now, B denotes the set of Byzantine vertices and C_B (which is function of B) denotes a subset of V (intuitively, this set gathers all vertices which may be

disturbed by Byzantine vertices).

Definition 13.19 (***C_B-correct vertex***)

*A vertex is C_B-*correct *if it is a correct vertex (*i.e. *not Byzantine) which not belongs to C_B.*

Definition 13.20 (***C_B-legitimate configuration***)

A configuration γ is C_B-legitimate for spec *if every C_B-correct vertex v is legitimate for* spec *(*i.e. *if* spec(v) *holds).*

Definition 13.21 (***(C_B, f)-topology-aware containment***)

A configuration γ_0 is (C_B, f)-topology-aware contained for specification spec *if, given at most f Byzantine vertices, in any execution $\sigma = (\gamma_0, \gamma_1)\ldots$, every configuration is C_B-legitimate and every C_B-correct vertex never changes its O-variables.*

The parameter C_B of Definition 13.21 refers to the containment area. Any vertex which belongs to this set may be infinitely often disturbed by Byzantine vertices. The parameter f refers explicitly to the number of Byzantine vertices.

Definition 13.22 (***Topology-aware strict stabilization***)

A distributed protocol is (C_B, f)-topology-aware strictly stabilizing for specification spec *if, given at most f Byzantine vertices, any execution contains a configuration that is (C_B, f)-topology-aware contained for* spec.

Note that, if B denotes the set of Byzantine vertices and

$$C_B = \left\{v \in V | \min_{b \in B}\{d(v, b)\} \leq c\right\}$$

then a (C_B, f)-topology-aware strictly stabilizing distributed protocol is a (c, f)-strictly stabilizing distributed protocol since C_B is then equals to the union of the c-neighborhood of Byzantine vertices. Then, the concept of topology-aware strict stabilization is a generalization of the strict stabilization. However, note that a TA strictly stabilizing protocol is stronger than a classical self-stabilizing protocol (that may never meet their specification in the presence of Byzantine vertices). The parameter C_B is introduced to quantify the strength of fault containment, we do not require each vertex to know the actual definition of the function.

Similarly to topology-aware strict stabilization, we can weaken the notion of strong stabilization using the notion of containment area. We present in the following the formal definition of this concept.

Definition 13.23 (***C_B-stable configuration***)

A configuration γ is C_B-stable if every C_B-correct vertex never changes the values of its O-variables as long as Byzantine vertices make no action.

Definition 13.24 (*C_B-TA disruption*)

A portion of execution $\delta = (\gamma_0, \gamma_1) \dots (\gamma_{t-1}, \gamma_t)$ $(t > 1)$ is a C_B-TA-disruption if and only if the followings hold:

1. *δ is finite;*
2. *δ contains at least one action of a C_B-correct vertex for changing the value of an O-variable;*
3. *γ_0 is C_B-legitimate for* spec *and C_B-stable; and*
4. *γ_t is the first configuration after γ_0 such that γ_t is C_B-legitimate for* spec *and C_B-stable.*

Definition 13.25 (*(t, k, C_B, f)-TA time contained configuration*)

A configuration γ_0 is (t, k, C_B, f)-TA time contained for spec *if given at most f Byzantine vertices, the following properties are satisfied:*

1. *γ_0 is C_B-legitimate for* spec *and C_B-stable;*
2. *every execution starting from γ_0 contains a C_B-legitimate configuration for* spec *after which the values of all the O-variables of C_B-correct vertices remain unchanged (even when Byzantine vertices make actions repeatedly and forever);*
3. *every execution starting from γ_0 contains at most t C_B-TA-disruptions; and*
4. *every execution starting from γ_0 contains at most k actions of changing the values of O-variables for each C_B-correct vertex.*

Definition 13.26 (*Topology-aware strong stabilization*)

A distributed protocol π is (t, C_B, f)-TA strongly stabilizing for specification spec *if and only if starting from any arbitrary configuration, every execution involving at most f Byzantine vertices contains a (t, k, C_B, f)-TA time contained configuration for* spec *that is reached after at most ℓ rounds. Parameters ℓ and k are respectively the (t, C_B, f)-stabilization time and the (t, C_B, f)-vertex disruption time of π.*

13.3.4 Discussion

Figure 13.1 sums up the respective constraints between strict, strong, topology-aware and self-stabilization. Recall that a scheme is more constrained that another if any distributed protocol that satisfies the first satisfies the second (see Section 4.3). For example, strict stabilization is more constrained than strong stabilization since any (c, f)-strictly stabilizing distributed protocol is a $(0, c, f)$-strongly stabilizing distributed protocol. In particular, note that TA strict stabilization and strong stabilization are not comparable. Indeed, both of them weaken constraints of strict

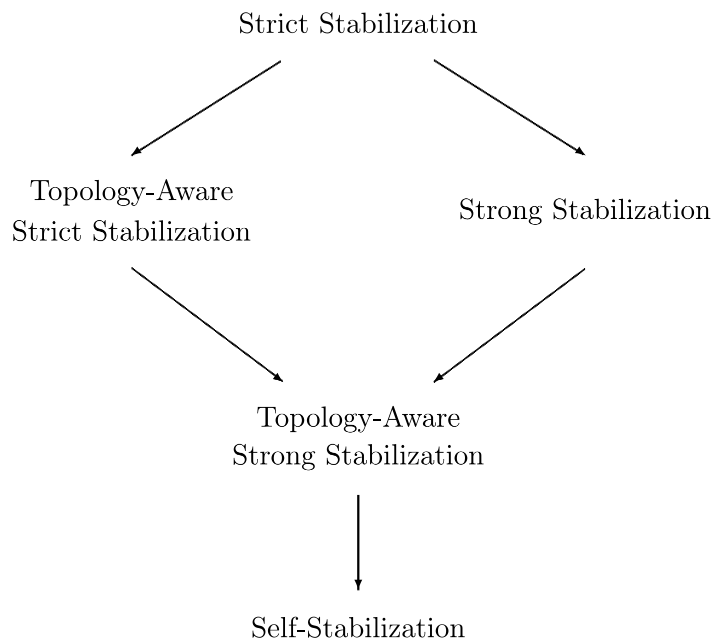

Figure 13.1: Summary of respective constraints on Byzantine containment schemes in self-stabilization. An arrow from a scheme to another means that the first is more constrained than the second.

stabilization but the first one removes constraints on space whereas the second removes constraints in time.

It is natural to conjecture that the more constrained is a scheme the more difficult is to provide a distributed protocol according to this scheme. Hence, we try to by-pass strict stabilization impossibility results by providing strongly and topology-aware stabilizing solutions.

Chapter 14
Two Case Studies

> The art of doing mathematics consists in finding that special case which contains all the germs of generality.
>
> David Hilbert

Contents

14.1 Spanning Tree without Constraints **180**
14.1.1 Strongly Stabilizing Distributed Protocol 181
14.1.2 Proof of Strong Stabilization 182
14.2 BFS Spanning Tree **185**
14.2.1 Impossibility of Strong Stabilization 186
14.2.2 Topology-Aware Stabilizing Solution 187
14.2.3 Proof of Topology-Aware Strict Stabilization 188
14.2.4 Proof of Topology-Aware Strong Stabilization 192
14.3 Summary **195**

Before studying the general case of maximum metric spanning tree construction for any maximizable metric in Chapter 15, we propose in this chapter two particular case studies: the spanning tree (without constraints) construction and the breadth-first search spanning tree construction. According to definitions of Section 13.1.2, these two spanning trees are indeed particular cases of the maximum metric spanning tree construction since it is sufficient to consider (maximizable) metrics $\mathcal{NC}$ and $\mathcal{BFS}$ respectively.

Contributions of this chapter are twofold and their detailed description follow.

Spanning tree construction: Section 14.1 is devoted to the spanning tree (without constraints) construction in systems subject to any transient and intermittent Byzantine fault pattern. The specification of this problem will be denoted by $spec_{NCT}$. We prove the existence of a $(t, 0, n-1)$-strongly stabilizing distributed protocol for $spec_{NCT}$ with a finite t. The containment radius of 0 is naturally optimal, that allows us to not consider topology-aware strong stabilization on this problem. Note that results of Chapter 15 imply that it is not interesting to consider topology-aware strict stabilization on this problem since the optimal containment area is $V \setminus \{r\}$.

BFS spanning tree construction: Section 14.2 focuses on the breadth first search (BFS) spanning tree construction in systems subject to any transient and intermittent Byzantine fault pattern. The specification of this problem will be denoted by $spec_{BFST}$. We prove the following set of results:

1. there exists no strongly stabilizing distributed protocol for $spec_{BFST}$ whatever is the considered containment radius and the maximum number of disruptions;
2. there exists a topology-aware strictly stabilizing distributed protocol for $spec_{BFST}$ with a containment area containing all correct vertices that are closer (or at equal distance) from a Byzantine vertex than the root; and
3. there exists a topology-aware strongly stabilizing distributed protocol for $spec_{BFST}$ with a containment area containing all correct vertices that are strictly closer from a Byzantine vertex than the root.

Note that these two containment areas are proved optimal by results of Chapter 15.

These two studies show us that all maximizable metrics do not have the same properties with respect to strong stabilization and topology aware stabilization, that motivate the general characterization provided in Chapter 15.

14.1 Spanning Tree without Constraints

In this section, we study our first particular case of maximum metric spanning tree construction using only the metric $\mathcal{NC}$ (defined in Section 13.1.2) that corresponds to a spanning tree without any supplementary constraints. In order to simplify the presentation of this section, we do not use directly Specification 13.1 with metric $\mathcal{NC}$ but we provide a simpler specification predicate that is equivalent.

In what follows, each vertex v has only two O-variables: a pointer to its parent in its tree ($prnt_v \in N_v \cup \{\bot\}$) and an integer which stores a distance ($dist_v \in \mathbb{N}$). It is indeed useless to store the metric value of each vertex since this latter is always equals to 0 by definition of the metric $\mathcal{NC}$. We can now specify the spanning tree (without supplementary constraints) in the following way.

Specification 14.1 (*Spanning tree construction $spec_{NCT}$*)

The specification predicate $spec_{NCT}(v)$ of the spanning tree construction for vertex v follows:

$$spec_{NCT}(v) : \begin{cases} prnt_v = \bot \text{ and } dist_v = 0 \text{ if } v \text{ is the root } r \\ prnt_v \in N_v \text{ and } dist_v = dist_{prnt_v} + 1 \text{ otherwise} \end{cases}$$

Notice that $spec_{NCT}$ requires that a spanning tree is constructed at any 0-legitimate configuration when no Byzantine vertex exists and that a spanning forest

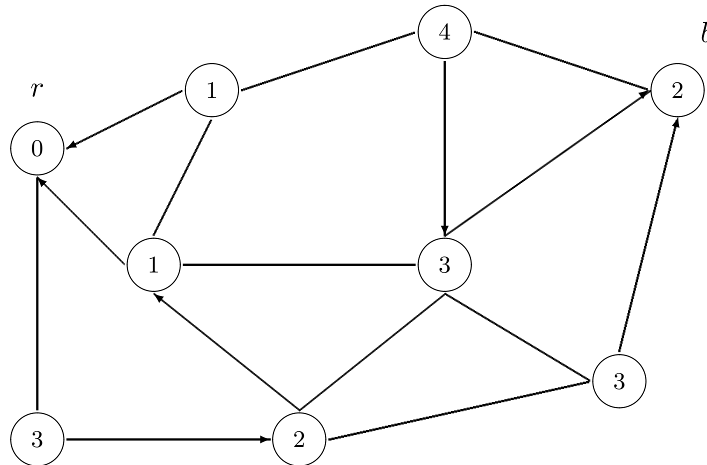

Figure 14.1: A 0-legitimate configuration for $spec_{NCT}$ (numbers denote the *dist* variable of vertices while the arrow attached to each vertex points the neighbor designated as its parent by the variable *prnt*). Note that r is the (real) root and b is a Byzantine vertex which acts as a (fake) root.

is constructed at any 0-legitimate configuration when Byzantine vertices exist. Figure 14.1 shows an example of 0-legitimate configuration with Byzantine vertices. Indeed, we can observe that any correct vertex satisfies $spec_{NCT}$ and that pointers from a spanning forest of the communication graph such that each root of this spanning forest is either the (real) root of the distributed system or a Byzantrine vertex.

The contribution of this section is to provide a $(t, 0, n-1)$-strongly stabilizing distributed protocol for $spec_{NCT}$ under the distributed strongly fair daemon for a finite natural number t. The containment radius of 0 is naturally optimal.

14.1.1 Strongly Stabilizing Distributed Protocol

In many self-stabilizing spanning tree construction distributed protocols (see the survey of [Gär03]), each vertex checks locally the consistence of its *dist* variable with respect to the one of its parent. When it detects an inconsistency, it changes its *prnt* variable in order to choose a "better" neighbor. The notion of "better" neighbor is based on the global desired property on the spanning tree (*e.g.* shortest path spanning tree, minimum degree spanning tree...).

When the system may contain Byzantine vertices, they may disturb their neighbors by providing alternatively "better" and "worse" states. The key idea of our distributed protocol $\mathcal{SSST}$ (for $\mathcal{S}$trongly $\mathcal{S}$tabilizing $\mathcal{S}$panning $\mathcal{T}$ree) to circumvent this kind of perturbation is the following: when a correct vertex detects a local inconsistency, it does not choose a "better" neighbor but it chooses another neighbor according to a round robin order (along the set of its neighbor).

Protocol 14.1 presents our strongly-stabilizing spanning tree construction distributed protocol $\mathcal{SSST}$ that can tolerate any number of Byzantine vertices other

than the root vertex (providing that the subset of correct vertices remains connected). These assumptions are necessary since a Byzantine root or a set of Byzantine vertices that disconnects the set of correct vertices may disturb all the tree infinitely often. Then, it is impossible to provide a (t, c, f)-strongly stabilizing protocol for any finite natural number t.

Note that $\mathcal{SSST}$ is designed for the distributed strongly fair daemon. The distributed protocol is composed of two rules. Only the root can execute the first one: $(\boldsymbol{R_r})$. This rule sets the root in a legitimate state if it is not the case. Non-root vertices may execute the other rule: $(\boldsymbol{R_v})$. The rule $(\boldsymbol{R_v})$ is executed when the state of a vertex is not consistent with the one of its parent. Its execution leads the vertex to choose a new parent (it choose the first greater neighbor of its current parent according to a round robin order defined on its set of neighbors) and to compute its local state in function of this new parent.

Protocol 14.1 $\mathcal{SSST}$: $(t, 0, n-1)$-strongly stabilizing distributed protocol for $spec_{NCT}$ for vertex v.

Constant

N_v: set of neighbors of v (ordered in a round robin fashion)

Variables

$prnt_v \in N_v \cup \{\bot\}$: parent of v in the current tree
$dist_v \in \mathbb{N}$: distance of v in the current tree

Functions

$next_v$: for any subset $A \subseteq N_v$, $next_v(A)$ returns the first element of A that is bigger than $prnt_v$ in a round-robin fashion and an arbitrary element of A if $prnt_v = \bot$

Rules

$(\boldsymbol{R_r}) :: (v = r) \wedge [(prnt_v \neq \bot) \vee (dist_v \neq 0)]$
$\quad\longrightarrow prnt_v := \bot;\ dist_v := 0$
$(\boldsymbol{R_v}) :: (v \neq r) \wedge [(prnt_v = \bot) \vee (dist_v \neq dist_{prnt_v} + 1)]$
$\quad\longrightarrow prnt_v := next_v(N_v);\ dist_v := dist_{prnt_v} + 1$

14.1.2 Proof of Strong Stabilization

This section provides the proof of the strong stabilization of $\mathcal{SSST}$. This proof is based on the fact that, once there exists a path (according to pointers) between a correct vertex and the real root, this one is preserved during the whole execution. In this way, we can prove that any correct vertex takes a finite number of steps during any execution (even if Byzantine vertices modify infinitely often their state). Then, the strong stabilization of $\mathcal{SSST}$ easily follows since the construction of the distributed protocol ensures us that a configuration in which no correct vertex is enabled is a 0-legitimate configuration for $spec_{NCT}$.

We denote by $\mathcal{LC}_{NCT}$ the following set of configurations:

$$\mathcal{LC}_{NCT} = \Big\{\gamma \in \Gamma \Big| (prnt_r = 0) \wedge (dist_r = 0) \wedge \big(\forall v \in V \setminus (B \cup \{r\}), (prnt_v \in N_v) \wedge (dist_v = dist_{prnt_v} + 1)\big)\Big\}$$

In other words, $\mathcal{LC}_{NCT}$ is the set of 0-legitimate configurations for $spec_{NCT}$. We interest now on properties of configurations of $\mathcal{LC}_{NCT}$.

Lemma 14.1

Any configuration of $\mathcal{LC}_{NCT}$ is 0-legitimate and 0-stable.

Proof: Let γ be a configuration of $\mathcal{LC}_{NCT}$. By definition of $\mathcal{LC}_{NCT}$, γ is 0-legitimate. Note that no correct vertex is enabled by $\mathcal{SSST}$ in γ. Consequently, no actions of $\mathcal{SSST}$ can be executed and we can deduce that γ is 0-stable. ■

Lemma 14.2

Given at most $n-1$ Byzantine vertices, for any initial configuration γ_0 and any execution $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots$ starting from γ_0 under the distributed strongly fair daemon, there exists a configuration γ_i such that $\gamma_i \in \mathcal{LC}_{NCT}$.

Proof: First, note that if all the correct vertices are disabled in a configuration γ, then γ belongs to $\mathcal{LC}_{NCT}$. Thus, it is sufficient to show that $\mathcal{SSST}$ eventually reaches a configuration γ_i in any execution (starting from any configuration) such that all the correct vertices are disabled in γ_i.

By contradiction, assume that there exists a correct vertex that is enabled infinitely often. Notice that once the root vertex r is activated, r becomes and remains disabled forever. From the assumption that all the correct vertices induce a connected communication subgraph, there exists two neighboring correct vertices u and v such that u becomes and remains disabled and v is enabled infinitely often. Consider execution after u becomes and remains disabled. Since the daemon is strongly fair, v executes its action infinitely often. Then, eventually v designates u as its parent. It follows that v never becomes enabled again unless u changes $dist_u$. Since u never becomes enabled, this leads to the contradiction. ■

From now, g' designates the communication subgraph induced by the set of all the correct vertices. Remember that g' is a connected communication graph by assumption.

Lemma 14.3

Any configuration in $\mathcal{LC}_{NCT}$ is a $(f \times deg(g)^{diam(g')}, deg(g)^{diam(g')}, 0, f)$-time contained configuration for $spec_{NCT}$.

Proof: Let γ_0 be a configuration of $\mathcal{LC}_{NCT}$ and $\sigma = (\gamma_0, \gamma_1)(\gamma_1, \gamma_2)\ldots$ be an execution starting from γ_0. First, we show that any 0-correct vertex v takes at most $deg(g)^{diam(g')}$ actions in σ. We prove this result by induction on the distance $dist(g', v, r)$ of v from the root in g'.

Induction basis ($dist(g', v, r) = 1$):

Let v be any vertex neighboring to the root r. Since γ_0 is a 0-legitimate configuration, $prnt_r = 0$ and $dist_r = 0$ hold at γ_0 and remain unchanged in σ. Thus, if $prnt_v = r$ and $dist_v = 1$ hold in a configuration γ, then v never changes $prnt_v$ or $dist_v$ in any execution starting from γ. Since $prnt_v = r$ and $dist_v = 1$ hold within the first $deg(g, v) - 1 \leq deg(g)$ actions of v, v can execute its action at most $deg(g)$ times in any execution starting from γ_0.

Induction step (with induction assumption):

Let v be any correct vertex $dist(g', v, r)$ hops away from the root r in g', and u be a correct neighbor of v that is $dist(g', v, r) - 1$ hops away from r in g' (this vertex exists by the assumption that g', the communication subgraph of correct vertices of g, is connected). From the induction assumption, u can execute its action at most $deg(g)^{dist(g',v,r)-1}$ times.

Assume that $prnt_v = u$ and $dist_v = dist_u + 1$ hold in a given configuration γ. We can observe that v is not enabled until u does not modify its state. Then, the round-robin order used for pointers modification allows us to deduce that v executes at most $deg(g, v) \leq deg(g)$ actions between two actions of u (or before the first action of u). By the induction assumption, u executes its action at most $deg(g)^{dist(g',v,r)-1}$ times. Thus, v can execute its action at most $deg(g) + deg(g) \times (deg(g)^{dist(g',v,r)-1}) = deg(g)^{dist(g',v,r)}$ times.

Consequently, any 0-correct vertex takes at most $deg(g)^{diam(g')}$ actions in σ.

We say that a Byzantine vertex b deceive a correct neighbor v in the action (γ, γ') if the state of b makes the guard of an action of v true in γ and if v executes this action in this action.

As a 0-disruption can be caused only by an action of a Byzantine vertex from a legitimate configuration, we can bound the number of 0-disruptions by counting the total number of times that correct vertices are deceived by neighboring Byzantine vertices.

If a 0-correct vertex v is deceived by a Byzantine neighbor b, it takes necessarily $deg(g, v)$ actions before being deceiving again by b (recall that we use a round-robin policy for $prnt_v$). As any 0-correct vertex v takes at most $deg(g)^{diam(g')}$ actions in σ, v can be deceived by a given Byzantine neighbor at most $deg(g)^{diam(g')-1}$ times. A Byzantine vertex can have at most $deg(g)$ neighboring correct vertices and thus can deceive correct vertices at most $deg(g) \times deg(g)^{diam(g')-1} = deg(g)^{diam(g')}$ times. We have at most f Byzantine vertices, so the total number of times that correct vertices are deceived by neighboring Byzantine vertices is $f \times deg(g)^{diam(g')}$.

Hence, the number of 0-disruption in σ is bounded by $f \times deg(g)^{diam(g')}$. It remains to show that any 0-disruption have a finite length to prove the result.

By contradiction, assume that there exists an infinite 0-disruption $\delta = (\gamma_i, \gamma_{i+1}) \ldots$ in σ. This implies that for all $j \geq i$, γ_j is not in $\mathcal{LC}_{NCT}$, that contradicts Lemma 14.2. Then, the result is proved. ■

Theorem 14.1

$\mathcal{SSST}$ is a $(f \times deg(g)^{diam(g')}, 0, f)$-strongly stabilizing distributed protocol for $spec_{NCT}$ under the distributed strongly fair daemon, where f is the number of Byzantine vertices and g' is the communication subgraph induced by the set of all the correct vertices.

Proof: From Lemmas 14.1 and 14.3, it is sufficient to show that $\mathcal{SSST}$ eventually reaches a configuration in $\mathcal{LC}_{NCT}$. Lemma 14.2 allows us to conclude. ■

14.2 BFS Spanning Tree

In this section, we study our second particular case of maximum metric spanning tree construction using metric $\mathcal{BFS}$ (defined in Section 13.1.2) that induces a breadth first search spanning tree (that is, the distance of each vertex to the root in the spanning tree is equal to the one in the original graph). As previously, we simplify the presentation of this section using an adapted specification of the problem.

In this section, each vertex v has only two O-variables: the first is $prnt_v \in N_v \cup \{\bot\}$ which is a pointer to the neighbor that is designated to be the parent of v in the BFS tree and the second is $level_v \in \{0, \ldots, diam(g)\}$ which stores the metric value of v in this tree. Indeed, it is useless to store the distance of the vertex to the root in the tree since it is equals to its metric value by definition of the metric $\mathcal{BFS}$. Then, we can specify the BFS spanning tree construction as follow.

Definition 14.1 ($\mathcal{BFS}$-path)

A path $(v_0, \ldots, v_k)$ $(k \geq 1)$ of g is a $\mathcal{BFS}$-path if and only if:

1. $prnt_{v_0} = \bot$, $level_{v_0} = 0$, *and* $v_0 \in B \cup \{r\}$;
2. $\forall i \in \{1, \ldots, k\}, prnt_{v_i} = v_{i-1}$ *and* $level_{v_i} = level_{v_{i-1}} + 1$; *and*
3. $\forall i \in \{1, \ldots, k\}, level_{v_{i-1}} = \underset{u \in N_v}{min}\{level_u\}$.

Specification 14.2 (*BFS spanning tree construction* $spec_{BFST}$)

The specification predicate $spec_{BFST}(v)$ of the BFS spanning tree construction for vertex v follows.

$$spec_{BFST}(v) : \begin{cases} prnt_v = \bot \text{ and } level_v = 0 \text{ if } v \text{ is the root } r \\ \text{there exists a } \mathcal{BFS}\text{-path } (v_0, \ldots, v_k) \text{ such that } v_k = v \text{ otherwise} \end{cases}$$

In the case where any vertex is correct, note that any 0-legitimate configuration for $spec_{BFST}$ contains a BFS spanning tree rooted to the real root. Otherwise, any C_B-legitimate configuration contains a BFS forest that spans at least the communication subgraph induced by $V \setminus C_B$ for any containment area $C_B \subseteq V$.

We present now contributions of this section. We prove first that there exists no strongly stabilizing solution for $spec_{BFST}$ for any containment radius (see Theorem 14.2). Then, we demonstrate that a classical self-stabilizing distributed protocol for BFS spanning tree construction known as the $min + 1$ protocol (see [HC92]) provides without significant changes some Byzantine containment properties. In more details, we prove in Theorems 14.3 and 14.4 that this distributed protocol is both (S_B, f)-TA strictly and (t, S_B^*, f)-TA strongly stabilizing where $f \leq n-1$, t is

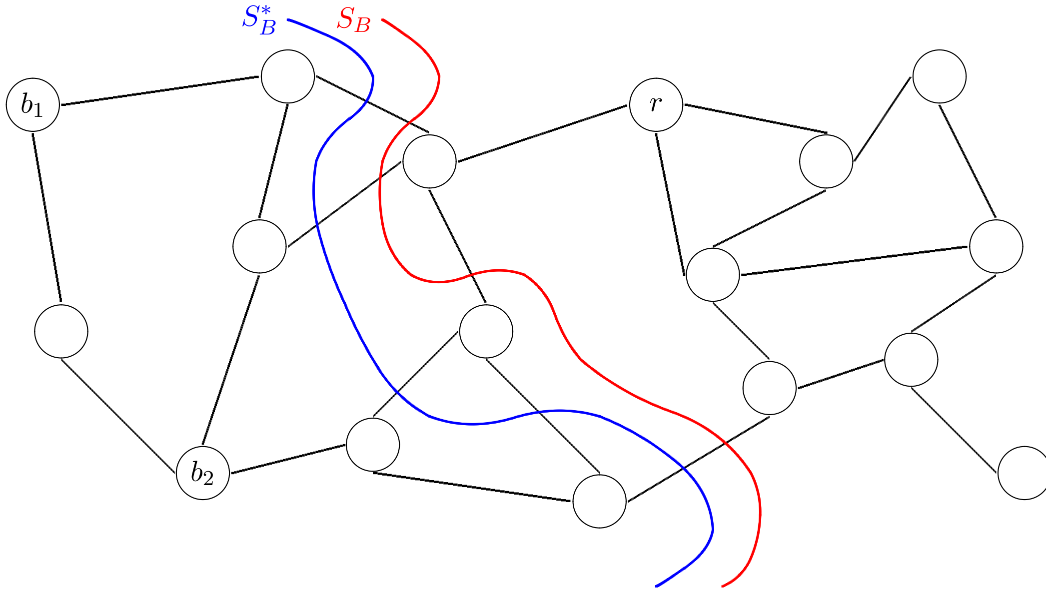

Figure 14.2: Example of containment areas for BFS spanning tree construction (vertices b_1 and b_2 are Byzantine and vertex r is the root).

finite, and

$$\begin{aligned} S_B &= \left\{ v \in V \middle| \min_{b \in B}\{dist(g,v,b)\} \leq dist(g,r,v) \right\} \\ S_B^* &= \left\{ v \in V \middle| \min_{b \in B}\{dist(g,v,b)\} < dist(g,r,v) \right\} \end{aligned}$$

Intuitively, S_B gathers the set of correct vertices that are closer (or at equals distance) from a Byzantine vertex than the root whereas S_B^* gathers the set of correct vertices that are strictly closer from a Byzantine vertex than the root. Figure 14.2 provides an example of these containment areas.

14.2.1 Impossibility of Strong Stabilization

We state first an impossibility result about strong stabilization for BFS spanning tree construction for any containment radius and any number of disruptions even with a single Byzantine failure. This result comes from the following observation. If a Byzantine vertex exhibits alternatively a behavior of a root vertex and of a correct vertex, it may lead any distributed protocol (that constructs a BFS spanning tree) to reconstruct a new spanning tree on the half of any path from the Byzantine vertex to the real root. The length of this path can be arbitrary and we deduce the impossibility to contain the effect of the Byzantine fault within any constant radius.

Theorem 14.2

Even under the central daemon, there exists no $(t, c, 1)$-strongly stabilizing distributed protocol for $spec_{BFST}$ where t and c are any (finite) natural numbers.

Proof: Let t and c be (finite) natural numbers. Assume that there exists a $(t, c, 1)$-strongly stabilizing distributed protocol π for $spec_{BFST}$ under the central daemon. Let $g = (V, E)$ be the following communication graph $V = \{v_0 = r, v_1, \ldots, v_{2c+2}, v_{2c+3} = b\}$ and $E = \{\{v_i, v_{i+1}\}, i \in \{0, \ldots, 2c+2\}\}$. Vertex v_0 is the real root and vertex b is a Byzantine one.

Assume that the initial configuration γ_0 of g satisfies: $level_r = level_b = 0$, $prnt_r = prnt_b = \perp$ and other variables of b (if any) are identical to those of r (see Figure 14.3). Assume now that b takes exactly the same actions as r (if any) immediately after r (note that $dist(g, r, b) > c$ and hence $level_r = 0$ and $prnt_r = \perp$ still hold by closure and then $level_b = 0$ and $prnt_b = \perp$ still hold too). Then, by symmetry of the execution and by convergence of π to $spec_{BFST}$, we can deduce that the system reaches in a finite time a configuration γ_1 (see Figure 14.3) in which: $\forall i \in \{1, \ldots, c+1\}, level_{v_i} = i$ and $prnt_{v_i} = v_{i-1}$ and $\forall i \in \{c+2, \ldots, 2c+2\}, level_{v_i} = 2c+3-i$ and $prnt_{v_i} = p_{i+1}$ (because this configuration is the only one in which all correct vertex v_i such that $d(g, v_i, b) > c$ satisfies $spec_{BFST}(v_i)$ when $level_r = level_b = 0$ and $prnt_r = prnt_b = \perp$). Note that γ_1 is 0-legitimate and 0-stable and *a fortiori* c-legitimate and c-stable.

Assume now that the Byzantine vertex acts as a correct vertex and executes correctly π. Then, by convergence of π in fault-free systems (remember that a $(t, c, 1)$-strongly stabilizing distributed protocol is a special case of self-stabilizing distributed protocol), we can deduce that the system reaches in a finite time a configuration γ_2 (see Figure 14.3) in which: $\forall i \in \{1, \ldots, 2c+3\}, level_{v_i} = i$ and $prnt_{v_i} = v_{i-1}$ (because this configuration is the only one in which all vertex v_i satisfies $spec_{BFST}(v_i)$). Note that the portion of execution between γ_1 and γ_2 contains at least one c-disruption (v_{c+2} is a c-correct vertex and modifies at least once its O-variables) and that γ_2 is 0-legitimate and 0-stable and *a fortiori* c-legitimate and c-stable.

Assume now that the Byzantine vertex b takes the following state: $level_b = 0$ and $prnt_b = \perp$. This action brings the system into configuration γ_3 (see Figure 14.3). From this configuration, we can repeat the execution we constructed from γ_0. By the same token, we obtain an execution of π which contains c-legitimate and c-stable configurations (see γ_1) and an infinite number of c-disruptions which contradicts the $(t, c, 1)$-strong stabilization of π. ■

14.2.2 Topology-Aware Stabilizing Solution

In this section, we present our distributed protocol for BFS spanning tree construction in systems subject to any transient and intermittent Byzantine fault pattern. This distributed protocol is borrowed from the one of [HC92] that is self-stabilizing for the BFS spanning tree construction. This distributed protocol is known as the $min + 1$ protocol. This name is due to the construction of the distributed protocol itself. Each vertex has two variables: one pointer to its parent in the tree and one level in this tree. The distributed protocol is reduced to the following rule: each vertex chooses as its parent the neighbor which has the smallest level (*min* part) and update its level in consequence (+1 part).

In the $min + 1$ protocol, as in many self-stabilizing spanning tree construction protocols, each vertex v checks locally the consistence of its $level_v$ variable with

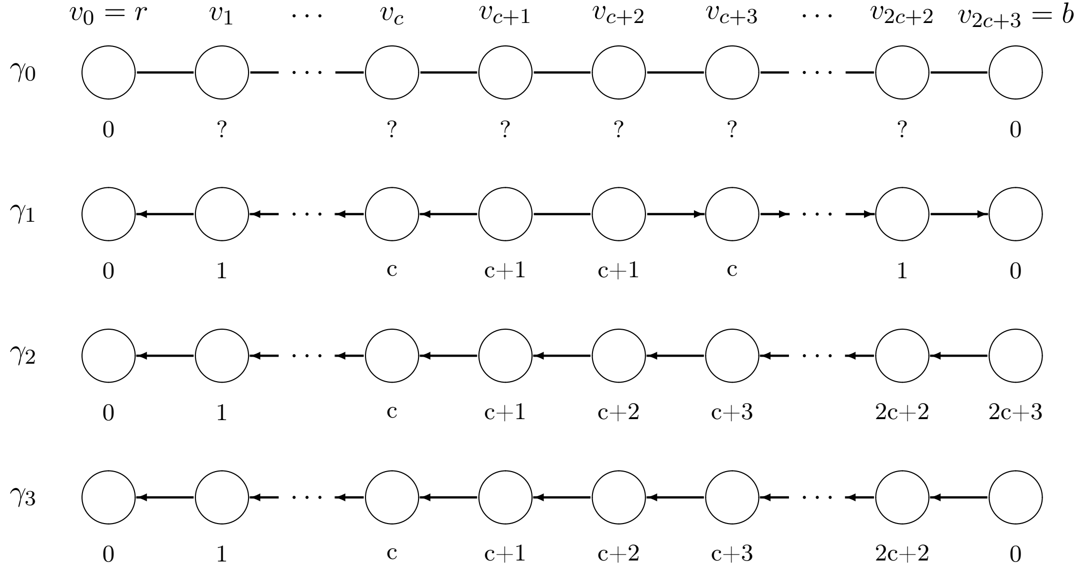

Figure 14.3: Configurations used in proof of Theorem 14.2.

respect to the one of its neighbors. When it detects an inconsistency, it changes its $prnt_v$ variable in order to choose a "better" neighbor. The notion of "better" neighbor is based on the global desired property on the tree (here, the BFS requirement implies to choose one neighbor with the minimum level). When the system may contain Byzantine vertices, they may disturb their neighbors by providing alternatively "better" and "worse" *level* values.

The $min + 1$ protocol chooses an arbitrary one of the "better" neighbors (that is, a neighbor with the minimal level). Actually this strategy allows us to achieve the (S_B, f)-TA strict stabilization but is not sufficient to achieve the (t, S_B^*, f)-TA strong stabilization. To achieve the (t, S_B^*, f)-TA strong stabilization, we must bring a slight modification to the distributed protocol: we choose a "better" neighbor with a round robin order (along the set of its neighbor) as in Section 14.1.

Protocol 14.2 presents our BFS spanning tree construction distributed protocol $\mathcal{SSBFS}$ (for $\mathcal{S}$trictly/strongly $\mathcal{S}$tabilizing $\mathcal{BFS}$) which is both (S_B, f)-TA strictly and (t, S_B^*, f)-TA strongly stabilizing for $spec_{BFST}$ (where $f \leq n-1$ and $t = n \times deg(g)$) providing that the root is never Byzantine.

In the following, we prove the $(S_B, n-1)$-TA strict stabilization and the $(n \times deg(g), S_B^*, n-1)$-TA strong stabilization of $\mathcal{SSBFS}$ respectively under the distributed weakly fair daemon and under the distributed strongly fair daemon.

14.2.3 Proof of Topology-Aware Strict Stabilization

The proof of the topology-aware strict stabilization of our distributed protocol is very similar to the one of the self-stabilization of the $min+1$ protocol. Indeed, we prove the convergence by induction on the distance of vertices to the root with the

Protocol 14.2 $\mathcal{SSBFS}$: $(S_B, n-1)$-TA strictly and $(t, S_B^*, n-1)$-TA strongly stabilizing distributed protocol for $spec_{BFST}$ for vertex v.

Constant

N_v: set of neighbors of v (ordered in a round robin fashion)

Variables

$prnt_v \in N_v \cup \{\bot\}$: parent of v in the current tree
$dist_v \in \{0, \ldots, diam(g)\}$: distance of v in the current tree

Functions

$next_v$: for any subset $A \subseteq N_v$, $next_v(A)$ returns the first element of A that is bigger than $prnt_v$ in a round-robin fashion and an arbitrary element of A if $prnt_v = \bot$

Rules

$(\boldsymbol{R_r}) :: (v = r) \wedge [(prnt_v \neq \bot) \vee (level_v \neq 0)]$
$\longrightarrow prnt_v := \bot;\ level_v := 0$

$(\boldsymbol{R_v}) :: (v \neq r) \wedge \Big[(prnt_v = \bot) \vee (level_v \neq level_{prnt_v} + 1) \vee \big(level_{prnt_v} \neq \min_{u \in N_v}\{level_u\}\big)\Big]$
$\longrightarrow prnt_v := next_v\Big(\big\{u \in N_v \big| level_u = \min_{w \in N_v}\{level_w\}\big\}\Big);$
$level_v := level_{prnt_v} + 1$

significant difference that this induction must take in account Byzantine vertices and consider only correct vertices that are closer from the root than from a Byzantine vertex. The closure of the set of S_B-legitimate configurations directly follows from the construction of the distributed protocol.

Given a configuration $\gamma \in \Gamma$ and a natural number $d \in \{0, \ldots, diam(g)\}$, let us define the following predicate:

$$I_d(\gamma) \equiv \forall v \in V, level_v \geq min\Big\{d, \min_{u \in B \cup \{r\}}\{dist(g, v, u)\}\Big\}$$

Lemma 14.4

For any natural number $d \in \{0, \ldots, diam(g)\}$, the predicate I_d is closed by actions of $\mathcal{SSBFS}$.

Proof: Let d be a natural number such that $d \in \{0, \ldots, diam(g)\}$. Let $\gamma \in \Gamma$ be a configuration such that $I_d(\gamma) = true$ and $\gamma' \in \Gamma$ be a configuration such that (γ, γ') is an action of $\mathcal{SSBFS}$.

If the root vertex $r \in Act(\gamma, \gamma')$ (respectively a Byzantine vertex $b \in Act(\gamma, \gamma')$), then we have $level_r = 0$ (respectively $level_b \geq 0$) in γ' by construction of $(\boldsymbol{R_r})$ (respectively by definition of $level_b$). Hence, $level_r \geq min\Big\{d, \min_{u \in B \cup \{r\}}\{dist(g, r, u)\}\Big\} = 0$ (respectively $level_b \geq min\Big\{d, \min_{u \in B \cup \{r\}}\{dist(g, b, u)\}\Big\} = 0$).

If a correct vertex $v \in Act(\gamma, \gamma')$ with $v \neq r$, then there exists a neighbor u of v which satisfies the following property in γ (since v is activated and $I_d(\gamma) = true$):

$$level_u = \min_{w \in N_v}\{level_w\} \geq min\Big\{d, \min_{w \in B \cup \{r\}}\{dist(g, u, w)\}\Big\}$$

Once, v is activated, we have: $level_v = level_u + 1$ in γ'. Let us define:

$$d' = \min_{w \in B \cup \{r\}} \{dist(g, v, w)\}$$

Then, we have: $\min_{w \in B \cup \{r\}} \{dist(g, u, w)\} \geq d - 1$ (otherwise, we have a contradiction with the fact that $d = \min_{w \in B \cup \{r\}} \{dist(g, u, w)\}$ and that v and u are neighbors). Consequently, γ' satisfies:

$$\begin{aligned} level_v = level_u + 1 &\geq min\Big\{d, \min_{w \in B \cup \{r\}} \{dist(g, u, w)\}\Big\} + 1 \\ &\geq min\{d, d' - 1\} + 1 \\ &\geq min\{d, d'\} \\ &\geq min\Big\{d, \min_{w \in B \cup \{r\}} \{dist(g, v, w)\}\Big\} \end{aligned}$$

We can deduce that $I_d(\gamma') = true$, that conclude the proof. ■

Let $\mathcal{LC}_{BFST}$ be the following set of configurations:

$$\mathcal{LC}_{BFST} = \Big\{\gamma \in \Gamma \Big| (\gamma \text{ is } S_B\text{-legitimate for } spec_{BFST}) \wedge (I_{diam(g)}(\gamma) = true)\Big\}$$

In other words, $\mathcal{LC}_{BFST}$ is the set of S_B-legitimate configurations for $spec_{BFST}$ such that any vertex has a level variable at least equals to its distance to the nearest vertex in $B \cup \{r\}$. We prove in the following interesting properties on configurations of $\mathcal{LC}_{BFST}$.

Lemma 14.5

Any configuration of $\mathcal{LC}_{BFST}$ is $(S_B, n-1)$-TA contained for $spec_{BFST}$.

Proof: Let γ be a configuration of $\mathcal{LC}_{BFST}$. By construction, γ is S_B-legitimate for $spec_{BFST}$.

In particular, the root vertex satisfies: $prnt_r = \bot$ and $level_r = 0$. By construction of $\mathcal{SSBFS}$, r is not enabled and then never modifies its O-variables (since the guard of the rule of r does not involve the state of its neighbors).

In the same way, any vertex $v \in V \setminus (S_B \cup \{r\})$ satisfies: $prnt_v \in N_v$, $level_v = level_{prnt_v} + 1$, and $level_{prnt_v} = \min_{u \in N_v} \{level_u\}$. Note that, as $v \in V \setminus (S_B \cup \{r\})$ and $spec_{BFST}(v)$ holds in γ, we have: $level_v = dist(g, v, r)$. Hence, vertex v is not enabled in γ. It remains so until none of its neighbors u modifies its $level_u$ variable to a value ℓ such that $\ell \leq level_v - 2$.

Assume that there exists an execution σ starting from γ in which a neighbor u of a vertex $v \in V \setminus (S_B \cup \{r\})$ modifies $level_u$ such that $level_u \leq level_v - 2$ (without loss of generality, assume that u is the first vertex to modify $level_u$ in such a way in σ). Note that $\min_{w \in B \cup \{r\}} \{dist(g, u, w)\} \geq dist(g, v, r) - 1$ (otherwise, we have a contradiction with the fact that $dist(g, v, r) = \min_{w \in B \cup \{r\}} \{dist(g, v, w)\}$ and that v

and u are neighbors). Hence, we have:

$$\begin{aligned}\min_{w \in B \cup \{r\}} \{dist(g,u,w)\} &\geq dist(g,v,r) - 1 \\ &> dist(g,v,r) - 2 \\ &> level_u\end{aligned}$$

This contradicts the closure of predicate $I_{diam(g)}$ established in Lemma 14.4.

Consequently, there exists no such execution and we can deduce that vertex v remains infinitely disabled and then never modifies its O-variables. This concludes the proof. ∎

Lemma 14.6

Starting from any configuration, any execution of $\mathcal{SSBFS}$ reaches a configuration of $\mathcal{LC}$ in a finite time under the distributed weakly fair daemon.

Proof: We first prove the following property by induction on $d \in \{0, \dots, diam(g)\}$:

($\mathcal{P}_d$): Starting from any configuration, any execution of $\mathcal{SSBFS}$ reaches a configuration γ such that $I_d(\gamma) = true$ and in which any vertex $v \notin S_B$ such that $dist(g,v,r) \leq d$ satisfies $spec_{BFST}(v)$ in a finite time under the distributed weakly fair daemon.

Initialization: $d = 0$.

Let γ be an arbitrary configuration. Then, it is obvious that $I_0(\gamma)$ is satisfied. If a vertex $v \notin S_B$ satisfies $dist(g,v,r) \leq 0$, then $v = r$. If v does not satisfy $spec_{BFST}(v)$ in γ, then v is continuously enabled in any execution starting from γ. Since the daemon is weakly fair, v is activated in a finite time and then v satisfies $spec_{BFST}(v)$ in a finite time. Then, we proved that ($\mathcal{P}_0$) holds.

Induction: $d \geq 1$ and $\mathcal{P}_{d-1}$ is true.

We know, by $\mathcal{P}_{d-1}$, that any execution of $\mathcal{SSBFS}$ under the distributed weakly fair daemon reaches a configuration γ such that $I_{d-1}(\gamma) = true$ and in which any vertex $v \notin S_B$ such that $dist(g,v,r) \leq d-1$ satisfies $spec_{BFST}(v)$.

Let us define

$$E_d = \Big\{ v \in V \big| \min_{u \in B \cup \{r\}} \{dist(g,v,u)\} \geq d \Big\}$$

Note that $I_{d-1}(\gamma)$ implies that $\forall v \in E_d, level_v \geq d-1$ (since $\forall v \in E_d$, $min\big\{d-1, \min_{u \in B \cup \{r\}} \{dist(g,v,u)\}\big\} = d-1$ by construction).

Note that any vertex $v \in E_d$ such that $level_v = d-1$ is enabled by $(\boldsymbol{R_v})$ since we have: $level_{prnt_v} \geq d-1$ (by $I_{d-1}(\gamma)$ and the fact that $prnt_v$ is a neighbor of v) and thus $level_v = d-1 < level_{prnt_v} + 1$. Moreover, this rule remains enabled until v is activated by closure of $I_{d-1}(\gamma)$ (see Lemma 14.4). As the daemon is weakly fair, we deduce that any vertex $v \in E_d$ such that $level_v = d-1$ is activated in a finite time in any execution starting from γ. Then, we can conclude that any execution starting from γ reaches in a finite time a configuration γ' such that $I_d(\gamma') = true$.

Let $v \notin S_B$ be a vertex such that $dist(g,r,v) = d$. We distinguish the following two cases:

Case 1: $spec_{BFST}(v)$ holds in γ' (and then $level_v = d$).
By closure of I_d, any configuration of any execution starting from γ' satisfies I_d. Moreover, v satisfies $dist(g,v,r) < \min_{u \in B}\{dist(g,v,u)\}$. Hence, there exists a $\mathcal{BFS}$-path from v to r. By construction, vertex v is then not enabled (remind that any neighbor u of v satisfies: $level_u \geq min\left\{d, \min_{w \in B \cup \{r\}}\{dist(g,u,w)\}\right\} \geq d$). In conclusion, v always satisfies $spec_{BFST}(v)$ in any execution starting from γ'.

Case 2: $spec_{BFST}(v)$ does not hold in γ'.
By construction of γ', we can split N_v into two sets S and $\bar{S}$ such that any vertex u of S satisfies $level_u = dist(g,r,u) = d-1$ and $spec_{BFST}(u)$ (and thus there exists a $\mathcal{BFS}$-path from u to r) and any vertex u of $\bar{S}$ satisfies $level_u \geq d$ (remind that $I_d(\gamma') = true$ and then $level_u \geq min\left\{d, \min_{w \in B \cup \{r\}}\{dist(g,u,w)\}\right\} \geq d$).
As $spec_{BFST}(v)$ does not hold in γ', we can deduce that v is enabled in γ'. As I_d is closed (by Lemma 14.4), we can deduce that v remains enabled. Since the daemon is weakly fair, we conclude that v is activated in a finite time in any execution starting from γ' and then $prnt_v$ is a vertex of S that implies that v satisfies $spec_{BFST}(v)$ in a finite time in any execution starting from γ'.

In conclusion, $\mathcal{P}_d$ is true, that ends the induction.

Then, it is easy to see that $\mathcal{P}_{diam(g)}$ implies the result. ■

Theorem 14.3

$\mathcal{SSBFS}$ is a $(S_B, n-1)$-TA strictly stabilizing distributed protocol for $spec_{BFST}$ under the distributed weakly fair daemon.

Proof: This result is a direct consequence of Lemmas 14.5 and 14.6. ■

14.2.4 Proof of Topology-Aware Strong Stabilization

This section is devoted to the proof of the topology-aware strong stabilization of $\mathcal{SSBFS}$. We describe here the main steps of this proof. Starting from a S_B-legitimate configuration (reached in a finite time by Theorem 14.3), we prove that any correct vertex of $S_B \setminus S_B^*$ takes only a finite number of steps in any execution (due to the fact that it joins in a finite number of step a path to the real root which is stable). Then, the result follows from the fact that a correct vertex of $S_B \setminus S_B^*$ cannot be never activated and invalidate $spec_{BFST}$ infinitely.

Let be $E_B = S_B \setminus S_B^*$. In other words, E_B is the set of vertices that are at equals distance to the root than to their nearest Byzantine vertex (*i.e.* E_B is the set of vertices v such that $dist(g,r,v) = \min_{b \in B}\{dist(g,v,b)\}$).

Lemma 14.7

If γ is a configuration of $\mathcal{LC}_{BFST}$, then any vertex $v \in E_B$ is activated at most $deg(g,v)$ times in any execution starting from γ.

Proof: Let γ be a configuration of $\mathcal{LC}_{BFST}$ and v a vertex of E_B. By construction, there exists a neighbor u of v such that $u \in V \setminus S_B$. Then, we know that $spec_{BFST}(u)$ holds in γ. By Lemma 14.5, we are ensured that $spec_{BFST}(u)$ remains satisfied in any configuration of any execution starting from γ. In particular, $level_u = dist(g, r, u)$. By closure of $I_{diam(g)}(\gamma)$ (established in Lemma 14.4), we know that $level_w \geq dist(g, r, u)$ for any neighbor w of v. Consequently, $level_u = \min_{w \in N_v}\{level_w\}$. This implies that, if $prnt_v = u$ and $level_v = level_u + 1$ in a configuration γ', then $spec_{BFST}(v)$ is satisfied and v takes no actions in any execution starting from γ'.

Then, the construction of the macro *next* implies that u is chosen as v's parent in at most $deg(g, v)$ actions of v. This implies the result. ■

Lemma 14.8

If γ is a configuration of $\mathcal{LC}_{BFST}$ and v is a vertex such that $v \in E_B$, then, for any execution σ starting from γ under the distributed strongly fair daemon, either

1. *v is activated in σ; or*
2. *there exists a configuration γ' of σ such that $spec_{BFST}(v)$ is always satisfied after γ'.*

Proof: Let γ be a configuration of $\mathcal{LC}_{BFST}$ and v be a vertex such that $v \in E_B$. By contradiction, assume that there exists an execution σ starting from γ such that (i) $spec_{BFST}(v)$ is infinitely often false in σ and (ii) v is never activated in σ.

For any configuration γ, let us denote by $P_v(\gamma) = (v, v_1 = prnt_v, v_2 = prnt_{v_1}, \ldots, v_k = prnt_{v_{k-1}}, p_v = prnt_{v_k})$ the maximal sequence of vertices following pointers $prnt$ (maximal means here that either $prnt_{p_v} = \perp$ or p_v is the first vertex such that $p_v = v_i$ for some $i \in \{1, \ldots, k\}$).

Let us study the following cases:

Case 1: $prnt_v \in V \setminus S_B$ in γ.
Since $\gamma \in \mathcal{LC}_{BFST}$, $prnt_v$ satisfies $spec_{BFST}(prnt_v)$ in γ and in any execution starting from γ (by Lemma 14.5). If v does not satisfy $spec_{BFST}(v)$ in γ, then we have $level_v \neq level_{prnt_v} + 1$ in γ. Then, v is continuously enabled in σ and we have a contradiction between assumption (ii) and the strong fairness of the scheduling. This implies that v satisfies $spec_{BFST}(v)$ in γ. The closure of $I_{diam(g)}$ (established in Lemma 14.4) ensures us that v is never enabled in any execution starting from γ. Hence, $spec_{BFST}(v)$ remains true in any execution starting from γ. This contradicts the assumption (i) on σ.

Case 2: $prnt_v \notin V \setminus S_B$ in γ.
By the assumption (i) on σ, we can deduce that there exists infinitely many configurations γ' such that a vertex of $P_v(\gamma')$ is enabled in γ'. By construction, the length of $P_v(\gamma')$ is finite for any configuration γ' and there exists only a finite number of vertices in the system. Consequently, there exists at least one vertex which is infinitely often enabled in σ. Since the daemon is strongly fair, we can conclude that there exists at least one vertex which is infinitely often activated in σ.

Let A_σ be the set of vertices which are infinitely often activated in σ. Note that $v \notin A_\sigma$ by assumption (ii) on σ. Let σ' be the suffix of σ which contains only activations of vertices of A_σ. Assume that σ' starts in configuration γ'. Let u be the first vertex of $P_v(\gamma')$ which belongs to A_σ (u exists since at least one vertex of P_v is enabled when $spec_{BFST}(v)$ is false). By construction, the prefix of $P_v(\gamma'')$ from v to u in any configuration γ'' of σ remains the same as the one of $P_v(\gamma')$. Let u' be the vertex such that $prnt_{u'} = u$ in σ' (u' exists since $v \neq u$ implies that the prefix of $P_v(\gamma')$ from v to u counts at least two vertices). As u is infinitely often activated and as any activation of u modifies the value of $level_u$ (it takes at least two different values in σ'), we can deduce that u' is infinitely often enabled in σ' (since the value of $level_{u'}$ is constant by construction of σ' and u). Since the daemon is strongly fair, u' is activated in a finite time in σ', that contradicts the construction of u.

In the two cases, we obtain a contradiction with the construction of σ, that prove the result. ■

Let $\mathcal{LC}^*_{BFST}$ be the following set of configurations:

$$\mathcal{LC}^*_{BFST} = \left\{\gamma \in \Gamma \middle| (\rho \textit{ is } S^*_B\textit{-legitimate for } spec_{BFST}) \wedge (I_{diam(g)}(\rho) = true)\right\}$$

In other words, $\mathcal{LC}^*_{BFST}$ is the set of S^*_B-legitimate configurations for $spec_{BFST}$ such that any vertex has a level variable at least equals to its distance to the nearest vertex in $B \cup \{r\}$. Note that, as $S^*_B \subseteq S_B$, we can deduce that $\mathcal{LC}^*_{BFST} \subseteq \mathcal{LC}_{BFST}$. Hence, properties of Lemmas 14.7 and 14.8 also applies to configurations of $\mathcal{LC}^*$.

Lemma 14.9

Any configuration of $\mathcal{LC}^*_{BFST}$ is $(n \times deg(g), deg(g), S^*_B, n-1)$-TA time contained for $spec_{BFST}$.

Proof: Let γ be a configuration of $\mathcal{LC}^*_{BFST}$. As $S^*_B \subseteq S_B$, we know by Lemma 14.5 that any vertex v of $V \setminus S_B$ satisfies $spec_{BFST}(v)$ and takes no action in any execution starting from γ. Let v be a vertex of E_B. By Lemmas 14.7 and 14.8, we know that v takes at most $deg(g, v)$ actions in any execution starting from γ. Moreover, we know that v satisfies $spec_{BFST}(v)$ after its last action (otherwise, we obtain a contradiction between the two lemmas). Hence, any vertex of E_B takes at most $deg(g)$ actions and then, there are at most $n \times deg(g)$ S^*_B-TA disruptions in any execution starting from γ. By definition of a TA time contained configuration, we obtain the result. ■

Lemma 14.10

Starting from any configuration, any execution of $\mathcal{SSBFS}$ reaches a configuration of $\mathcal{LC}^*_{BFST}$ in a finite time under the distributed strongly fair daemon.

Proof: Let γ be an arbitrary configuration. We know by Lemma 14.6 that any execution starting from γ reaches in a finite time a configuration γ' of $\mathcal{LC}_{BFST}$ under the distributed weakly fair daemon and *a fortiori* under the distributed strongly fair daemon.

Let v be a vertex of E_B. By Lemmas 14.7 and 14.8, we know that v takes at most $deg(g, v)$ actions in any execution starting from γ'. Moreover, we know that v satisfies $spec_{BFST}(v)$ after its last action (otherwise, we obtain a contradiction between the two lemmas). This implies that any execution starting from γ' reaches (under the distributed strongly fair daemon) a configuration γ'' such that any vertex v of E_B satisfies $spec_{BFST}(v)$. It is easy to see that $\gamma'' \in \mathcal{LC}^*_{BFST}$, that ends the proof. ■

Theorem 14.4

$\mathcal{SSBFS}$ is a $(n \times deg(g), S^*_B, n-1)$-TA strongly stabilizing distributed protocol for $spec_{BFST}$ under the distributed strongly fair daemon.

Proof: This result is a direct consequence of Lemmas 14.9 and 14.10. ■

14.3 Summary

This chapter proves that all maximizable metrics does not implies similar results for Byzantine containment in self-stabilization since:

1. Theorem 14.1 proves that there exits a $(f \times deg(g)^{diam(g')}, 0, f)$-strongly stabilizing distributed protocol for maximum metric spanning tree construction with respect to $\mathcal{NC}$;
2. Theorem 14.2 shows that there exists no $(t, c, 1)$-strongly stabilizing distributed protocol for maximum metric spanning tree construction with respect to $\mathcal{BFS}$; and
3. Theorems 14.3 and 14.4 demonstrate respectively that there exists a $(S_B, n-1)$-TA strictly stabilizing and a $(n \times deg(g), S^*_B, n-1)$-TA strongly stabilizing distributed protocol for maximum metric spanning tree construction with respect to $\mathcal{BFS}$.

The aim of the following chapter is first to generalize results about topology-aware stabilization to any maximizable metric and then to characterize the subset of maximizable that admits a strongly stabilizing solution. Note that we also prove the optimality of containment areas of $\mathcal{SSBFS}$ for the BFS spanning tree construction.

Chapter 15

General Case

I realize that I'm generalizing here, but as is often the case when I generalize, I don't care.

Dave Barry

Contents

15.1 Topology-Aware Stabilizing Solution **200**
15.1.1 Distributed Protocol 201
15.1.2 Proof of Topology-Aware Strict Stabilization 204
15.1.3 Proof of Topology-Aware Strong Stabilization 211
15.2 Optimality of Containment Areas **215**
15.2.1 Topology-Aware Strict Stabilization 215
15.2.2 Topology-Aware Strong Stabilization 217
15.3 Strong Stabilization **219**
15.4 Summary **225**

In this chapter, we deal with the maximum metric spanning tree construction for any maximizable metric. In Chapter 14, we proved that all maximizable metric are not equivalent with respect to Byzantine containment. Indeed, some of them (like $\mathcal{NC}$) allow strong stabilization with a bounded containment radius (see Section 14.1) whereas some others (like $\mathcal{BFS}$) forbid strong stabilization for any finite containment radius but admit topology-aware stabilizing solutions (see Section 14.2).

We generalize now these results. Detailed contributions of this chapter follow.

1. Section 15.1 presents a distributed protocol and proves that this distributed protocol is topology-aware strictly and strongly stabilizing for some containment areas that generalizes those presented in Chapter 14 for BFS spanning tree construction.
2. We prove in Section 15.2 that the containment areas provided by the aforementioned distributed protocol are optimal (this result proves also the claim that the distributed protocol $\mathcal{SSBFS}$ of Chapter 14 performed the optimal Byzantine containment).
3. Finally, we provide in Section 15.3 a general result about strong stabilization. This result is a fully characterization of the set of maximizable metrics that allow the existence of a strongly stabilizing solution to the maximum metric spanning tree construction with a finite containment radius (this result also characterizes the optimal containment radius).

We need to discuss the specification of the problem and to bring some definitions before the presentation of these contributions.

Specification The specification of the maximum metric spanning tree presented in Section 13.1.2 is the more natural but, unfortunately, we did not succeed to provide all our results with it. Indeed, we can prove the topology-aware strict stabilization of our distributed protocol for $spec_{MMT}$ but we must consider a slight different specification to prove its topology-aware strong stabilization.

In order to keep the consistency of results presented in this chapter, we choose to use only this new specification of the problem. This specification differs from $spec_{MMT}$ only on the constraints on the variable *dist* (that stores the distance to the root in $spec_{MMT}$). We require now that this variable on vertex v stores the distance from v to the first vertex of the $\mathcal{M}$-path that have the same *level* value than v. We formally state this new specification in the following.

We define the legal distance of a vertex v with respect to one of its neighbors u (denoted $legal_dist(v,u)$) in the following way:

$$\forall v \in V, \forall u \in N_v, legal_dist(v,u) = \begin{cases} dist_u + 1 \text{ if } level_v = level_u \\ 0 \text{ otherwise} \end{cases}$$

We introduce now a slight variant of the $\mathcal{M}$-path (see Definition 13.10).

Definition 15.1 (*$\mathcal{M}$-improved path*)

Given an assigned metric $\mathcal{M} = (M, W, mr, met, \prec, wf)$ over a communication graph g, a path $(v_0, \dots, v_k)$ $(k \geq 1)$ of g is a $\mathcal{M}$-improved path if and only if:

1. $prnt_{v_0} = \bot$, $level_{v_0} = mr$, $dist_{v_0} = 0$, *and* $v_0 \in B \cup \{r\}$;
2. $\forall i \in \{1, \dots, k\}$, $prnt_{v_i} = v_{i-1}$, $level_{v_i} = met(level_{v_{i-1}}, w_{v_i, v_{i-1}})$, *and* $dist_{v_i} = legal_dist(v_i, v_{i-1})$;
3. $\forall i \in \{1, \dots, k\}$, $met(level_{v_{i-1}}, w_{v_i, v_{i-1}}) = \max_{\prec, u \in N_v} \{met(level_u, w_{v_i, u})\}$; *and*
4. $level_{v_k} = max_met(g, v_k, v_0)$.

We can now provide our new specification of the maximum metric spanning tree construction using only $\mathcal{M}$-improved paths instead of $\mathcal{M}$-paths.

Specification 15.1 (*Maximum metric spanning tree construction $spec_{IMMT}$*)

The specification predicate $spec_{IMMT}(v)$ of the maximum metric tree construction with respect to a maximizable metric $\mathcal{M}$ for vertex v follows:

$$spec_{IMMT}(v): \begin{cases} prnt_v = \bot, \ level_v = mr, \ \text{and } dist_v = 0 \text{ if } v \text{ is the root } r \\ \text{there exists a } \mathcal{M}\text{-improved path } (v_0, \dots, v_k) \text{ such that } v_k = v \text{ otherwise} \end{cases}$$

Note that, for any containment area S_B, any S_B-legitimate configuration for $spec_{IMMT}$ implies the existence of a maximum metric spanning forest that spans $V \setminus S_B$ exactly in the same way as in a S_B-legitimate configuration for $spec_{MMT}$. The only difference between these two configurations is the value of *dist* variables. Hence, we allow ourselves to present only results with respect to $spec_{IMMT}$.

Definitions We introduce here some new definitions to characterize some important properties of maximizable metrics that are used in the following.

A strictly decreasing metric is a particular case of a bounded metric since, for any metric value, the application of the metric operator either strictly decrease it or does not modify it (for any edge weight). A fixed point is a particular metric value such that the application of the metric never modifies it (for any edge weight). Formal definitions follow.

Definition 15.2 (*Strictly decreasing metric*)

A metric $\mathcal{M} = (M, W, mr, met, \prec)$ is strictly decreasing if, for any metric value $m \in M$, the following property holds: either $\forall w \in W, met(m, w) \prec m$ or $\forall w \in W, met(m, w) = m$.

Definition 15.3 (*Fixed point*)

A metric value m is a fixed point of a metric $\mathcal{M} = (M, W, mr, met, \prec)$ if $m \in M$ and if for any value $w \in W$, we have: $met(m, w) = m$.

Given an assigned metric over a communication graph, a used metric value is a metric value that is the metric of a rooted path of the communication graph (either on the root or on a Byzantine vertex). More formally,

Definition 15.4 (*Set of used metric values*)

Given an assigned metric $\mathcal{AM} = (M, W, met, mr, \prec, wf)$ over a communication graph g, the set of used metric values of $\mathcal{AM}$ is defined as:

$$M(g) = \{m \in M | \exists v \in V, (max_met(g, v, r) = m) \vee (\exists b \in B, max_met(g, v, b) = m)\}$$

Note that for any communication graph g and any assigned metric $\mathcal{AM} = (M, W, met, mr, \prec, wf)$ over g, we have: $M(g) \subseteq M$.

Containment areas We define now the containment areas used in this chapter. Note that they generalize those presented in Section 14.2 for the BFS spanning tree construction to any maximizable metric $\mathcal{M} = (M, W, mr, met, \prec)$. First, the containment area for topology-aware strict stabilization is the following:

$$S_B = \left\{ v \in V \setminus B \middle| max_met(g, v, r) \preceq \max_{b \in B}{}_{\prec} \{max_met(g, v, b)\} \right\} \setminus \{r\}$$

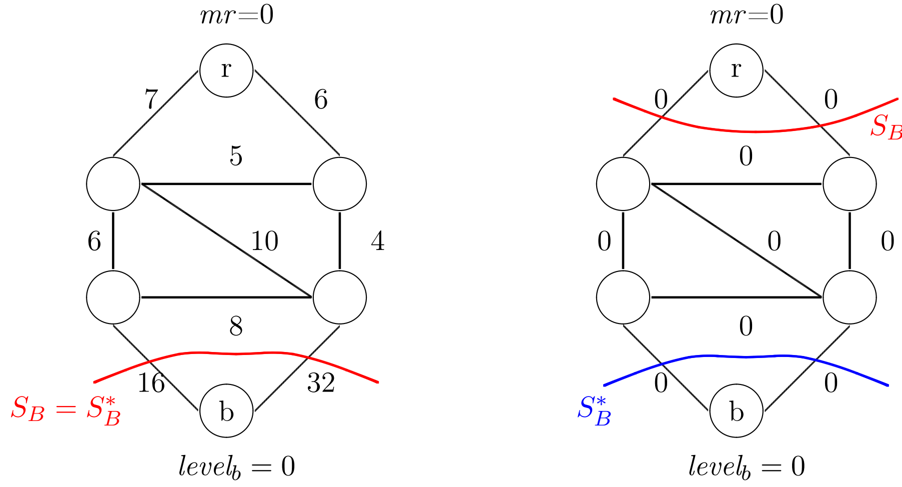

Figure 15.1: Examples of containment areas for $\mathcal{SP}$.

Intuitively, S_B gathers the set of correct vertices that are closer or at equals distance (according to $\mathcal{M}$) to a Byzantine vertex than the root. Then, we can define the containment area for topology-aware strong stabilization:

$$S^*_B = \left\{ v \in V \setminus B \middle| max_met(g, v, r) \prec \max_{b \in B}{}_{\prec} \{max_met(g, v, b)\} \right\}$$

Intuitively, S^*_B gathers the set of corrects vertices that are strictly closer (according to $\mathcal{M}$) to a Byzantine vertex than the root. Note that we assume for the sake of clarity that $V \setminus S^*_B$ induces a connected communication subgraph. If it is not the case, then S^*_B is extended to include all vertices belonging to connected communication subgraphs of $V \setminus S^*_B$ that not include r. Figures from 15.1 to 15.3 provide some examples of containment areas S_B and S_B with respect to maximizable metrics presented in Section 13.1.2.

15.1 Topology-Aware Stabilizing Solution

This section aims to present a distributed protocol for maximum metric spanning tree construction with respect to any maximizable metric in systems subject to any transient and intermittent Byzantine fault pattern. Section 15.1.1 presents the distributed protocol while Sections 15.1.2 and 15.1.3 prove respectively its $(S_B, n-1)$-TA strict stabilization and its $(t, S^*_B, n-1)$-TA strong stabilization.

Note that we prove the optimality of these containment areas in Section 15.2.

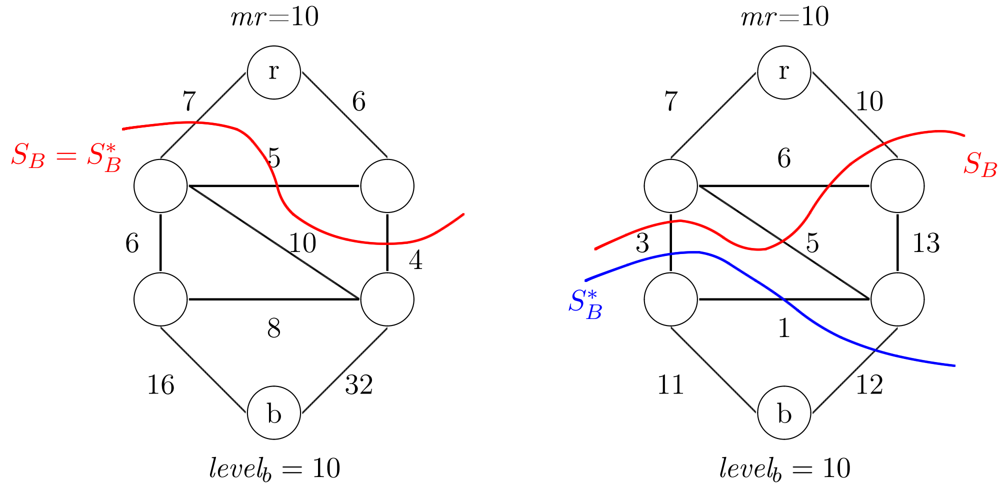

Figure 15.2: Examples of containment areas for $\mathcal{F}$.

15.1.1 Distributed Protocol

A self-stabilizing distributed protocol for maximum metric spanning tree construction with respect to any maximizable metric has been proposed by [GS99]. In this distributed protocol, any vertex try to maximize its *level* variable in the tree by choosing as its parent (*prnt* variable) the neighbor that provide the best metric value. Using this strategy, the arbitrary initial configuration may lead to the formation of cycles. The key idea of this distributed protocol is to use the *dist* variable (upper bounded by a given constant D) to detect and break cycles of vertices which has the same (incorrect) maximum metric. The choice of the constant D is obviously capital for the self-stabilization of the distributed protocol. Gouda and Schneider proved that their distributed protocol is self-stabilizing if D is an upper bound on the length of the longest path of the desired tree.

A natural way to provide a topology-aware stabilizing solution to the maximum metric spanning tree construction is then to adapt the idea of round robin choice over neighbors presented in Chapter 14 to the distributed protocol of [GS99]. It is possible to prove that this strategy is sufficient to perform the $(S_B, n-1)$-TA strict stabilization. Unfortunately, this strategy is not suitable for topology-aware strong stabilization.

Indeed, an execution of the distributed protocol of [GS99] may be subject to an infinite number of S_B^*-disruptions due to the following fact: a Byzantine vertex can independently lie about its *level* and its *dist* variable. For example, a Byzantine vertex can provide a *level* equals to mr and a *dist* arbitrarily large. In this way, it may lead a correct vertex of $S_B \setminus S_B^*$ to have a *dist* variable equals to $D-1$ such that no other correct vertex can choose it as its parent (this rule is necessary to break cycle) but it cannot modify its state (this rule is only enabled when *dist* is equals to D). Then, this vertex may always prevent some of its neighbors to join a

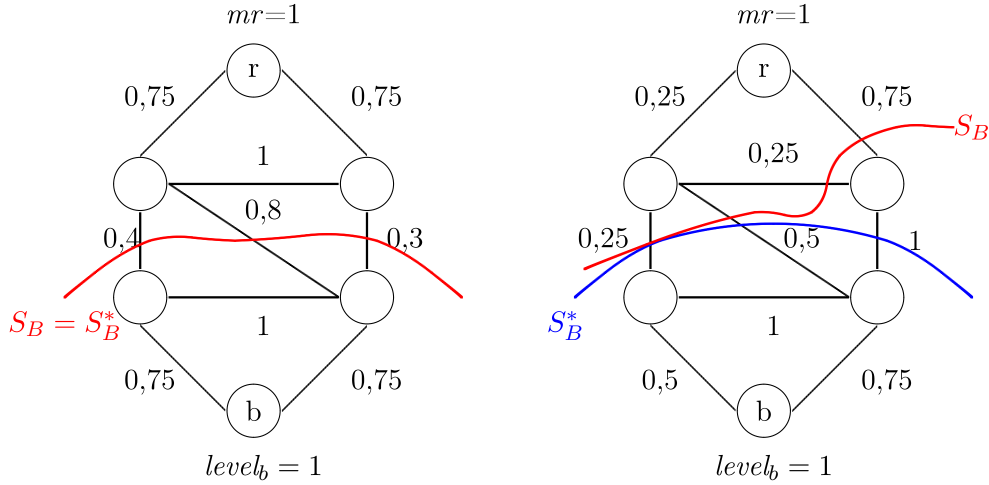

Figure 15.3: Examples of containment areas for $\mathcal{R}$.

$\mathcal{M}$-path connected to the root and hence allow another Byzantine vertex to perform an infinite number of disruptions.

In contrast, we want to provide a distributed protocol that is simultaneously $(S_B, n-1)$-TA strictly stabilizing and $(t, S_B^*, n-1)$-TA strongly stabilizing for maximum metric spanning tree construction. To perform this goal, our distributed protocol needs a supplementary assumption on the assignation of the considered maximizable metric over the communication graph.

We assume that we always have $|M(g)| \geq 2$ (the necessity of this assumption is explained below). Nevertheless, note that the contrary case ($|M(g)| = 1$) is possible if and only if the assigned maximizable metric is equivalent to $\mathcal{NC}$. As the distributed protocol presented in Section 14.1 performs $(t, 0, n-1)$-strong stabilization with a finite t for this metric, we can achieve the $(t, S_B^*, n-1)$-TA strong stabilization when $|M(g)| = 1$ (since this implies that $S_B^* = \emptyset$). In this way, this assumption does not weaken the possibility result.

We already said that the distributed protocol of [GS99] is not suitable for our purposes but our distributed protocol borrows fundamental strategy from it. Indeed, we use almost the same ideas with the two following exceptions: (i) we ensure a fair selection along the set of neighbor with a round-robin order for the *prnt* variable (as in the two distributed protocols presented in Chapter 14) and (ii) we modify the management of the *dist* variable to avoid executions exhibiting an infinite number of S_B^*-disruptions.

In order to contain the effect of Byzantine vertices on *dist* variables, each vertex that has a *level* different from the one of its parent in the tree sets its *dist* variable to 0. In this way, a Byzantine vertex modifying its *dist* variable can only affect correct vertices that have the same *level*. Consequently, in the case where $|M(g)| \geq 2$, we are ensured that correct vertices of $S_B \setminus S_B^*$ cannot keep a *dist* variable equals or

Protocol 15.1 $\mathcal{SSMMT}$: $(S_B, n-1)$-TA strictly and $(t, S_B^*, n-1)$-TA strongly stabilizing distributed protocol for $spec_{IMMT}$ for vertex v.

Constants:

N_v: set of neighbors of v (ordered in a round robin fashion)

D: upper bound of the number of vertices in an elementary path

Variables:

$prnt_v \in \begin{cases} \{\bot\} \text{ if } v = r \\ N_v \text{ if } v \neq r \end{cases}$: parent of v in the current tree

$level_v \in M$: metric of v in the current tree

$dist_v \in \{0, \dots, D\}$: distance of v in the current tree

Functions:

$next_v$: for any subset $A \subseteq N_v$, $next_v(A)$ returns the first element of A that is bigger than $prnt_v$ in a round-robin fashion and an arbitrary element of A if $prnt_v = \bot$

$current_dist_v() = \begin{cases} 0 \text{ if } level_{prnt_v} \neq level_v \\ min\{dist_{prnt_v} + 1, D\} \text{ if } level_{prnt_v} = level_v \end{cases}$

Rules:

$(\boldsymbol{R_r}) :: (v = r) \wedge ((level_v \neq mr) \vee (dist_v \neq 0))$
$\longrightarrow level_v := mr;\ dist_v := 0$

$(\boldsymbol{R_1}) :: (v \neq r) \wedge \left[(dist_v < current_dist_v()) \vee (level_v \neq met(level_{prnt_v}, w_{v,prnt_v}))\right]$
$\longrightarrow level_v := met(level_{prnt_v}, w_{v,prnt_v});\ dist_v := current_dist_v()$

$(\boldsymbol{R_2}) :: (v \neq r) \wedge \left[(dist_v = D) \vee (dist_v > current_dist_v())\right] \wedge (\exists u \in N_v, dist_u < D-1)$
$\longrightarrow prnt_v := next_v\big(\{u \in N_v | dist_v < D-1\}\big);$
$level_v := met(level_{prnt_v}, w_{v,prnt_v});\ dist_v := current_dist_v()$

$(\boldsymbol{R_3}) :: (v \neq r) \wedge (\exists u \in N_v, (dist_u < D-1) \wedge (level_v \prec met(level_u, w_{u,v})))$
$\longrightarrow prnt_v := next_v\big(\{u \in N_v | (level_u < D-1) \wedge$
$(met(level_u, w_{u,v}) = \max_{\prec\, q \in N_v / level_q < D-1} \{met(level_q, w_{q,v})\}\}\big);$
$level_v := met(level_{prnt_v}, w_{prnt_v,v});\ dist_v := current_dist_v()$

greater than $D-1$ infinitely. Hence, a correct vertex of $S_B \setminus S_B^*$ cannot be disturbed infinitely often without joining a $\mathcal{M}$-path connected to the root.

We can see that the assumption $|M(g)| \geq 2$ is essential to perform the topology-aware strong stabilization. Indeed, in the case where $|M(g)| = 1$, Byzantine vertices can play exactly the scenario described above (in this case, our distributed protocol is equivalent to the one of [GS99]).

The second modification we bring to the management of the *dist* variable follows. When a vertex has an inconsistent *dist* variable with its parent, we allow it only to increase its *dist* variable. If the vertex needs to decrease its *dist* variable (when it has a strictly greater distance than its parent), then the vertex must change its parent. This rule allows us to bound the maximal number of actions of any vertex between two modifications of its parent (a Byzantine vertex cannot lead a correct one to infinitely often increase and decrease its distance without modifying its pointer).

Our protocol, $\mathcal{SSMMT}$ (for $\mathcal{S}$trictly/strongly $\mathcal{S}$tabilizing $\mathcal{M}$aximum $\mathcal{M}$etric $\mathcal{T}$ree), is formally described in Protocol 15.1.

15.1.2 Proof of Topology-Aware Strict Stabilization

This section is devoted to the proof of the $(S_B, n-1)$-TA strict stabilization of $\mathcal{SSMMT}$ under the distributed weakly fair daemon (see Theorem 15.1). The main lines of this proof are similar to the one of Theorem 14.3 (proof of the topology-aware strict stabilization of our BFS spanning tree construction) but we must adapt it to take in account the fact that several vertices along a path could have the same maximal metric with respect to the root (or to a Byzantine vertex) that is not the case previously since the $\mathcal{BFS}$ metric is strictly decreasing. This new difficulty renders the proof quite technical but this proof remains an induction proof with respect to the maximal metric of each correct vertex.

We must first prove some lemmas. From now, we consider that $\mathcal{M} = (M, W, mr, met, \prec)$ is a maximizable metric assigned over our communication graph $g = (V, E)$ by the weight function wf. First, we provide a useful property about $\mathcal{M}$.

Lemma 15.1

For any vertex $v \in V$, we have:

$$\forall u \in N_v, met\Big(\max_{\prec, p \in B \cup \{r\}} \{max_met(g, u, p)\}, w_{u,v}\Big) \preceq \max_{\prec, p \in B \cup \{r\}} \{max_met(g, v, p)\}$$

Proof: By contradiction, assume that there exists a neighbor u of a vertex v such that:

$$\max_{\prec, p \in B \cup \{r\}} \{max_met(g, v, p)\} \prec met\Big(\max_{\prec, p \in B \cup \{r\}} \{max_met(g, u, p)\}, w_{u,v}\Big)$$

Let $q \in B \cup \{r\}$ be one of the vertices such that $\max_{\prec, p \in B \cup \{r\}} \{max_met(g, u, p)\} = max_met(g, u, q)$. Then, the construction of q allows us to deduce that:

$$\max_{\prec, p \in B \cup \{r\}} \{max_met(g, v, p)\} \prec met(max_met(g, u, q), w_{u,v})$$

Since we have $met(max_met(g, u, q), w_{u,v}) \preceq max_met(g, v, q)$, we conclude that:

$$\max_{\prec, p \in B \cup \{r\}} \{max_met(g, v, p)\} \prec max_met(g, v, q)$$

This contradicts the fact that $q \in B \cup \{r\}$ and shows us the result. ■

Given a configuration $\gamma \in \Gamma$ and a metric value $m \in M$, let us define the following predicate:

$$IM_m(\gamma) \equiv \forall v \in V, level_v \preceq \max_{\prec}\Big\{m, \max_{\prec, u \in B \cup \{r\}} \{max_met(g, v, u)\}\Big\}$$

Lemma 15.2

For any metric value $m \in M$, the predicate IM_m is closed by actions of $\mathcal{SSMMT}$.

Proof: Let m be a metric value ($m \in M$). Let $\gamma \in \Gamma$ be a configuration such that $IM_m(\gamma) = true$ and $\gamma' \in \Gamma$ be a configuration such that (γ, γ') is an action of $\mathcal{SSMMT}$.

If the root vertex $r \in Act(\gamma, \gamma')$ (respectively a Byzantine vertex $b \in Act(\gamma, \gamma')$), then we have $level_r = mr$ (respectively $level_b \preceq mr$) in γ' by construction of **(R_r)** (respectively by definition of $level_b$). Hence, we have:

$$level_r \preceq max_{\prec}\Big\{m, \max_{\prec\, u \in B \cup \{r\}} \{max_met(g, r, u)\}\Big\} = mr$$
$$level_b \preceq max_{\prec}\Big\{m, \max_{\prec\, u \in B \cup \{r\}} \{max_met(g, b, u)\}\Big\} = mr$$

If a correct vertex $v \in Act(\gamma, \gamma')$ with $v \neq r$, then there exists a neighbor p of v such that $level_p \preceq max_{\prec}\Big\{m, \max_{\prec\, u \in B \cup \{r\}} \{max_met(g, p, u)\}\Big\}$ in γ (since $IM_m(\gamma) = true$) and $prnt_v = p$ and $level_v = met(level_p, w_{v,p})$ in γ' (since v is activated during this action).

If we apply Lemma 15.1 to met and to neighbor p, we obtain the following property:

$$met\Big(\max_{\prec\, u \in B \cup \{r\}} \{max_met(g, p, u)\}, w_{v,p}\Big) \preceq \max_{\prec\, u \in B \cup \{r\}} \{max_met(g, v, u)\}$$

Consequently, we obtain that $level_v = met(level_p, w_{v,p})$ in γ'. The monotonicity of $\mathcal{M}$ allows us to deduce

$$\begin{aligned} level_v &\preceq met\Big(max_{\prec}\{m, \max_{\prec\, u \in B \cup \{r\}} \{max_met(g, p, u)\}\}, w_{v,p}\Big) \\ &\preceq max_{\prec}\Big\{met(m, w_{v,p}), met\big(\max_{\prec\, u \in B \cup \{r\}} \{max_met(g, p, u)\}, w_{v,p}\big)\Big\} \end{aligned}$$

As $met(m, w_{v,p}) \preceq m$, we can conclude that:

$$level_v \preceq max_{\prec}\Big\{m, \max_{\prec\, u \in B \cup \{r\}} \{max_met(g, v, u)\}\Big\}$$

We can deduce that $IM_m(\gamma') = true$, that concludes the proof. ∎

Given an assigned metric over a communication graph g, we can observe that the set of used metrics value $M(g)$ is finite and that we can label elements of $M(g)$ by $m_0 = mr, m_1, \ldots, m_k$ in a way such that $\forall i \in \{0, \ldots, k-1\}, m_{i+1} \prec m_i$.

We introduce the following notations:

$$\begin{aligned} \forall m_i \in M, \quad & P_{m_i} = \Big\{v \in V \setminus S_B \Big| max_met(g, v, r) = m_i\Big\} \\ \forall m_i \in M, \quad & V_{m_i} = \bigcup_{j=0}^{i} P_{m_j} \\ \forall m_i \in M, \quad & I_{m_i} = \Big\{v \in V \Big| \max_{\prec\, u \in B \cup \{r\}} \{max_met(g, v, u)\} \prec m_i\Big\} \\ \forall m_i \in M, \quad & \mathcal{LC}_{m_i} = \Big\{\gamma \in \Gamma \Big| (\forall v \in V_{m_i}, spec_{IMMT}(v)) \wedge (IM_{m_i}(\gamma))\Big\} \\ & \mathcal{LC}_{IMMT} = \mathcal{LC}_{m_k} \end{aligned}$$

Lemma 15.3

For any $m_i \in M(g)$, the set $\mathcal{LC}_{m_i}$ is closed by actions of $\mathcal{SSMMT}$.

Proof: Let m_i be a metric value from $M(g)$ and γ be a configuration of $\mathcal{LC}_{m_i}$. By construction, any vertex $v \in V_{m_i}$ satisfies $spec_{IMMT}(v)$ in γ.

In particular, the root vertex satisfies: $prnt_r = \perp$, $level_r = mr$, and $dist_r = 0$. By construction of $\mathcal{SSMMT}$, r is not enabled and then never modifies its O-variables (since the guard of the rule of r does not involve the state of its neighbors).

In the same way, any vertex $v \in V_{m_i}$ satisfies: $prnt_v \in N_v$, $level_v = met(level_{prnt_v}, w_{prnt_v,v})$, $dist_v = legal_dist_{prnt_v}$, and $level_v = \max_{\prec, u \in N_v}\{met(level_u, w_{u,v})\}$. Note that, as $v \in V_{m_i}$ and $spec_{IMMT}(v)$ holds in γ, we have: $level_v = max_met(g, v, r) = \max_{\prec, p \in B \cup \{r\}}\{max_met(g, v, p)\}$ and $dist_v \leq D - 1$ by construction of D. Hence, vertex v is not enabled by $\mathcal{SSMMT}$ in γ.

Assume that there exists a vertex $v \in V_{m_i}$ that is activated during an action (γ', γ'') in an execution starting from γ (without loss of generality, assume that v is the first vertex of $v \in V_{m_i}$ that is activated in this execution). Then, we know that $v \neq r$. This activation implies that a neighbor $u \notin V_{m_i}$ (since v is the first vertex of V_{m_i} to be activated) of v modified its *level* variable to a metric value $m \in M$ such that $level_v \prec met(m, w_{u,v})$ in γ' (note that O-variables of v and of $prnt_v$ remain consistent since v is the first vertex to be activated in this execution).

Hence, we have $level_v = \max_{\prec, p \in B \cup \{r\}}\{max_met(g, v, p)\} = max_met(g, v, r)$ (since $spec_{IMMT}(v)$ holds), $level_v \prec met(m, w_{u,v})$ (since u causes an action of v), and $m_i \preceq level_v$ (since $v \in V_{m_i}$ and $level_v = max_met(g, v, r)$). Moreover, the closure of IM_{m_i} (established in Lemma 15.2) ensures us that:

$$m = level_u \preceq \max_{\prec}\left\{m_i, \max_{\prec, p \in B \cup \{r\}}\{max_met(g, u, p)\}\right\}$$

Let us study the two following cases:

Case 1: $\max_{\prec}\left\{m_i, \max_{\prec, p \in B \cup \{r\}}\{max_met(g, u, p)\}\right\} = m_i$.

We have then $m \preceq m_i$. As the boundedness of $\mathcal{M}$ ensures that $met(m, w_{u,v}) \preceq m$, we can conclude that $level_v \prec met(m, w_{u,v}) \preceq m \preceq m_i \preceq level_v$, that is absurd.

Case 2: $\max_{\prec}\left\{m_i, \max_{\prec, p \in B \cup \{r\}}\{max_met(g, u, p)\}\right\} = \max_{\prec, p \in B \cup \{r\}}\{max_met(g, u, p)\}$.

We have then $m \preceq \max_{\prec, p \in B \cup \{r\}}\{max_met(g, u, p)\}$. By monotonicity of $\mathcal{M}$, we can deduce that $met(m, w_{u,v}) \preceq met(\max_{\prec, p \in B \cup \{r\}}\{max_met(g, u, p)\}, w_{u,v})$. Consequently, we obtain that:

$$\max_{\prec, p \in B \cup \{r\}}\{max_met(g, v, p)\} \prec met(\max_{\prec, p \in B \cup \{r\}}\{max_met(g, u, p)\}, w_{u,v})$$

This is contradictory with the result of Lemma 15.1.

In conclusion, no vertex $v \in V_{m_i}$ is activated in any execution starting from γ and then always satisfies $spec_{IMMT}(v)$. Then, the closure of IM_{m_i} (established in Lemma 15.2) concludes the proof. ■

Lemma 15.4

Any configuration of $\mathcal{LC}_{IMMT}$ is $(S_B, n-1)$-TA contained for $spec_{IMMT}$.

Proof: This is a direct application of the Lemma 15.3 to $\mathcal{LC}_{IMMT} = \mathcal{LC}_{m_k}$. ■

Lemma 15.5

Starting from any configuration of Γ, any execution of $\mathcal{SSMMT}$ under the distributed weakly fair daemon reaches in a finite time a configuration of $\mathcal{LC}_{mr}$.

Proof: Let γ be an arbitrary configuration. Then, it is obvious that $IM_{mr}(\gamma)$ is satisfied. By closure of IM_{mr} (proved in Lemma 15.2), we know that IM_{mr} remains satisfied in any execution starting from γ.

If r does not satisfy $spec_{IMMT}(r)$ in γ, then r is continuously enabled. Since the daemon is weakly fair, r is activated in a finite time and then r satisfies $spec_{IMMT}(r)$ in a finite time. Denote by γ' the first configuration in which $spec_{IMMT}(r)$ holds. Note that r is never activated in any execution starting from γ'.

The boundedness of $\mathcal{M}$ implies that P_{mr} induces a connected subsystem. If $P_{mr} = \{r\}$, then we proved that $\gamma' \in \mathcal{LC}_{mr}$ and we have the result.

Otherwise ($P_{mr} \neq \{r\}$), observe that, for any configuration of an execution starting from γ', if all vertices of P_{mr} are not enabled, then any vertex v of P_{mr} satisfies $spec_{IMMT}(v)$. Assume now that there exists an execution σ starting from γ' in which some vertices of P_{mr} are infinitely often activated. By construction, at least one of these vertices (note it v) has a neighbor u which is activated only a finite number of times in σ (recall that P_{mr} induces a connected subsystem and that r is not activated in σ). After u takes its last action of σ, we can observe that $level_u = mr$ and $dist_u < D - 1$ (otherwise, u is activated in a finite time that contradicts its construction).

As v can execute consequently $(\boldsymbol{R_1})$ only a finite number of times (since the incrementation of $dist_v$ is bounded by D), we can deduce that v executes $(\boldsymbol{R_2})$ or $(\boldsymbol{R_3})$ infinitely often in σ. In both cases, u belongs to the set which is the parameter of function $next_v$. By the fairness of this function, we can deduce that $prnt_v = u$ in a finite time in σ. Then, the construction of u implies that v is never enabled in the sequel of σ. This is contradictory with the construction of σ.

Consequently, any execution starting from γ' reaches in a finite time a configuration such that all vertices of P_{mr} are not enabled. We can deduce that this configuration belongs to $\mathcal{LC}_{mr}$, that ends the proof. ■

Lemma 15.6

For any $m_i \in M(g)$ and for any configuration $\gamma \in \mathcal{LC}_{m_i}$, any execution of $\mathcal{SSMMT}$ starting from γ under the distributed weakly fair daemon reaches in a finite time a configuration such that:

$$\forall v \in I_{m_i}, level_v = m_i \Rightarrow dist_v = D$$

Proof: Let m_i be an arbitrary metric value of $M(g)$ and γ_0 be an arbitrary configuration of $\mathcal{LC}_{m_i}$. Let $\sigma = (\gamma_0, \gamma_1) \dots$ be an execution starting from γ_0.

Note that γ_0 satisfies IM_{m_i} by construction. Hence, we have $\forall v \in I_{m_i}, level_v \preceq m_i$. The closure of IM_{m_i} (proved in Lemma 15.2) ensures us that this property is satisfied in any configuration of σ.

If any vertex $v \in I_{m_i}$ satisfies $level_v \prec m_i$ in γ_0, then the result is obvious. Otherwise, we define the following variant function. For any configuration γ_j of σ, we denote by A_j the set of vertices v of I_{m_i} such that $level_v = m_i$ in γ_j. Then, we define $f(\gamma_j) = \underset{v \in A_j}{min}\{dist_v\}$. We will prove the result by showing that there exists an integer k such that $f(\gamma_k) = D$.

First, if a vertex v joins A_j (that is, $v \notin A_{j-1}$ but $v \in A_j$), then it takes a *dist* value greater or equals to $f(\gamma_{j-1})+1$ by construction of $\mathcal{SSMMT}$. We can deduce that any vertex that joins A_j does not decrease f. Moreover, the construction of $\mathcal{SSMMT}$ implies that a vertex v such that $v \in A_j$ and $v \in A_{j+1}$ cannot decrease its *dist* value in the action (γ_j, γ_{j+1}).

Then, consider for a given configuration γ_j a vertex $v \in A_j$ such that $dist_v = f(\gamma_j) < D$. We claim that v is enabled by $\mathcal{SSMMT}$ in γ_j and that the execution of the enabled rule either increases strictly $dist_v$ or removes v from A_{j+1}. To prove this claim, we distinguish the following cases:

Case 1: $level_v = met(level_{prnt_v}, w_{v,prnt_v})$

The fact that $v \in I_{m_i}$, the boundedness of $\mathcal{M}$ and the closure of IM_{m_i} (established in Lemma 15.2) imply that $prnt_v \in A_j$ (and, hence that $level_{prnt_v} = m_i$). Then, by construction of $f(\gamma_j)$, we know that $dist_{prnt_v} \geq f(\gamma_j) = dist_v$. Hence, we have $dist_v < dist_{prnt_v} + 1$ in γ_j. Then, v is enabled by $(\boldsymbol{R_1})$ in γ_j and $dist_v$ increases of at least 1 during the action (γ_j, γ_{j+1}) if this rule is executed.

Case 2: $level_v \neq met(level_{prnt_v}, w_{v,prnt_v})$

Assume that v is activated by $(\boldsymbol{R_2})$ or $(\boldsymbol{R_3})$ during the action (γ_j, γ_{j+1}). If v does not belong to A_{j+1} (if $level_v \neq m_i$ in γ_{j+1}), the claim is satisfied. In the contrary case (v belongs to A_{j+1}), we know that $level_v = m_i$ in γ_{j+1}. The boundedness of $\mathcal{M}$ and the closure of IM_{m_i} (established in Lemma 15.2) imply that $level_{prnt_v} = m_i$ in γ_{j+1}. We can conclude that $dist_v$ increases of at least 1 during the action (γ_j, γ_{j+1}) since the new parent of v has a distance greater than $f(\gamma_j)$ by construction of A_{j+1}.

Otherwise, we know that the rule $(\boldsymbol{R_1})$ is enabled for v in γ_j. If this rule is executed during the action (γ_j, γ_{j+1}), one of the two following sub cases appears.

Case 2.1: $met(level_{prnt_v}, w_{v,prnt_v}) \prec m_i$ in γ_j.

Then, v does not belong to A_{j+1} by definition.

Case 2.2: $met(level_{prnt_v}, w_{v,prnt_v}) = m_i$ in γ_j.

Remind that the closure of IM_{m_i} (established in Lemma 15.2) implies then that $level_{prnt_v} = m_i$. By construction of $f(\gamma_j)$, we have $dist_{prnt_v} \geq f(\gamma_j)$ in γ_j. Then, we can see that $dist_v$ increases of at least 1 during the action (γ_j, γ_{j+1}).

In all cases, v is enabled (at least by $(\boldsymbol{R_1})$) in γ_j and the execution of the enabled rule either increases strictly $dist_v$ or removes v from A_{j+1}.

As I_{m_i} is finite and the daemon is weakly fair, we can deduce that f increases in a finite time in any execution starting from γ_j. By repeating the argument at most D times, we can deduce that σ contains a configuration γ_k such that $f(\gamma_k) = D$, that shows the result. ■

Lemma 15.7

For any $m_i \in M(g)$ and for any configuration $\gamma \in \mathcal{LC}_{m_i}$ such that $\forall v \in I_{m_i}, level_v = m_i \Rightarrow dist_v = D$, any execution of $\mathcal{SSMMT}$ starting from γ under the distributed weakly fair daemon reaches in a finite time a configuration such that:

$$\forall v \in I_{m_i}, level_v \prec m_i$$

Proof: Let $m_i \in M(g)$ be an arbitrary metric value and γ_0 be a configuration of $\mathcal{LC}_{m_i}$ such that $\forall v \in I_{m_i}, level_v = m_i \Rightarrow dist_v = D$. Let $\sigma = (\gamma_0, \gamma_1) \ldots$ be an arbitrary execution starting from γ_0.

For any configuration γ_j of σ, let us denote $E_{\gamma_j} = \{v \in I_{m_i} | level_v = m_i\}$. By the closure of IM_{m_i} (that holds by definition in γ_0) established in Lemma 15.2, we obtain the result if there exists a configuration γ_j of σ such that $E_{\gamma_j} = \emptyset$.

If there exist some vertices $v \in I_{m_i} \setminus E_{\gamma_0}$ (and hence $level_v \prec m_i$) such that $prnt_v \in E_{\gamma_0}$ and $met(level_{prnt_v}, w_{v,prnt_v}) = m_i$ in γ_0, then we can observe that these vertices are continuously enabled by $(\boldsymbol{R_1})$. As the daemon is weakly fair, v executes this rule in a finite time and then, $level_v = m_i$ and $dist_v = D$. In other words, v joins E_{γ_ℓ} for a given integer ℓ. We can conclude that there exists an integer k such that the following property $(\boldsymbol{P})$ holds: for any $v \in I_{m_i} \setminus E_{\gamma_0}$, either $prnt_v \notin E_{\gamma_k}$ or $met(level_{prnt_v}, w_{v,prnt_v}) \prec m_i$.

Then, we prove that, for any integer $j \geq k$, we have $E_{\gamma_{j+1}} \subseteq E_{\gamma_j}$. For the sake of contradiction, assume that there exists an integer $j \geq k$ and a vertex $v \in I_{m_i}$ such that $v \in E_{\gamma_{j+1}}$ and $v \notin E_{\gamma_j}$. Without loss of generality, assume that j is the smallest integer that satisfies these properties. Let us study the following cases:

Case 1: v executes $(\boldsymbol{R_1})$ during the action (γ_j, γ_{j+1}).
Note that the property $(\boldsymbol{P})$ still holds in γ_j by the construction of j. Hence, we know that $prnt_v \notin E_{\gamma_j}$ in γ_j. But in this case, we have: $level_{prnt_v} \prec m_i$. The boundedness of $\mathcal{M}$ implies that $level_v = met(level_{prnt_v}, w_{v,prnt_v}) \prec m_i$ in γ_{j+1} that contradicts the fact that $v \in E_{\gamma_{j+1}}$.

Case 2: v executes either $(\boldsymbol{R_2})$ or $(\boldsymbol{R_3})$ during the action (γ_j, γ_{j+1}).
That implies that v chooses a new parent which has a distance smaller than $D-1$ in γ_j. This implies that this new parent does not belongs to E_{γ_j}. Then, we have $level_{prnt_v} \prec m_i$. The boundedness of $\mathcal{M}$ implies that $level_v = met(level_{prnt_v}, w_{v,prnt_v}) \prec m_i$ in γ_{j+1} that contradicts the fact that $v \in E_{\gamma_{j+1}}$.

In the two cases, our claim is satisfied. In other words, there exists a point of the execution (namely γ_k) afterwards the set E cannot grow (this implies that, if a vertex leaves the set E, it is a definitive leaving).

Assume now that there exists an action (γ_j, γ_{j+1}) (with $j \geq k$) such that a vertex $v \in E_{\gamma_j}$ is activated. Observe that the closure of IM_{m_i} (established in Lemma 15.2) implies that v can not be activated by the rule $(\boldsymbol{R_3})$. If v activates $(\boldsymbol{R_1})$ during this action, then v modifies its *level* during this action (otherwise, we have a contradiction with the fact that $level_{prnt_v} = m_i \Rightarrow dist_v = D$). The closure of IM_{m_i} implies that v leaves the set E during this action. If v activates $(\boldsymbol{R_2})$ during this action, then v chooses a new parent which has a distance smaller than $D-1$ in γ_j. This implies that this new parent does not belongs to E_{γ_j}. Then, we have $level_{prnt_v} \prec m_i$. The boundedness of $\mathcal{M}$ implies that $level_v \prec m_i$ in γ_{j+1}.

In other words, if a vertex of E_{γ_j} is activated during the action (γ_j, γ_{j+1}), then it satisfies $v \notin E_{\gamma_{j+1}}$.

Finally, observe that the construction of $\mathcal{SSMMT}$ and the construction of the bound D ensures us that any vertex $v \in I_{m_i}$ such that $dist_v = D$ is activated in a finite time. In conclusion, we obtain that there exists an integer j such that $E_{\gamma_j} = \emptyset$, that implies the result. ∎

Lemma 15.8

For any $m_i \in M(g)$ and for any configuration of $\mathcal{LC}_{m_i}$, any execution of $\mathcal{SSMMT}$ starting from γ under the distributed weakly fair daemon reaches in a finite time a configuration such that $IM_{m_{i+1}}$ holds.

Proof: This result is a direct consequence of Lemmas 15.6 and 15.7. ∎

Lemma 15.9

For any $m_i \in M(g)$ and for any configuration $\gamma \in \mathcal{LC}_{m_i}$, any execution of $\mathcal{SSMMT}$ starting from γ under the distributed weakly fair daemon reaches in a finite time a configuration of $\mathcal{LC}_{m_{i+1}}$.

Proof: Let m_i be a metric value of $M(g)$ and γ be an arbitrary configuration of $\mathcal{LC}_{m_i}$. We know by Lemma 15.8 that any execution starting from γ reaches in a finite time a configuration γ' such that $IM_{m_{i+1}}$ holds. By closure of $IM_{m_{i+1}}$ and of $\mathcal{LC}_{m_i}$ (established respectively in Lemma 15.2 and 15.3), we know that any configuration of any execution starting from γ' belongs to $\mathcal{LC}_{m_i}$ and satisfies $IM_{m_{i+1}}$.

We know that $V_{m_i} \neq \emptyset$ since $r \in V_{m_i}$ for any $i \geq 0$. Remind that $V_{m_{i+1}}$ is connected by the boundedness of $\mathcal{M}$. Then, we know that there exists at least one vertex p of $P_{m_{i+1}}$ which has a neighbor q in V_{m_i} such that $max_met(g, p, r) = met(max_met(g, q, r), w_{p,q})$. Moreover, Lemma 15.3 ensures us that any vertex of V_{m_i} is not activated in any execution starting from γ'.

Observe that, for any configuration of any execution starting from γ', if any vertex of $P_{m_{i+1}}$ is not enabled, then all vertices v of $P_{m_{i+1}}$ satisfy $spec_{IMMT}(v)$. Assume now that there exists an execution σ starting from γ' in which some vertices of $P_{m_{i+1}}$ are infinitely often activated. By construction, at least one of these vertices (note it v) has a neighbor u such that $max_met(g, v, r) = met(max_met(g, u, r), w_{v,u})$ which takes only a finite number of actions in σ (recall the construction of p). After u takes its last action of σ, we can observe that $level_u = max_met(g, u, r)$ and $dist_u < D - 1$ (otherwise, u is activated in a finite time that contradicts its construction).

As v can execute consequently $\boldsymbol{(R_1)}$ only a finite number of times (since the incrementation of $dist_v$ is bounded by D), we can deduce that v executes $\boldsymbol{(R_2)}$ or $\boldsymbol{(R_3)}$ infinitely often. In both cases, u belongs to the set which is the parameter of function $next_v$ (remind that $IM_{m_{i+1}}$ is satisfied and that u has the better possible metric among v's neighbors). By the construction of this function, we can deduce that $prnt_v = u$ in a finite time in σ. Then, the construction of u implies that v is never enabled in the sequel of σ. This is contradictory with the construction of σ.

Consequently, any execution starting from γ' reaches in a finite time a configuration such that all vertices of $P_{m_{i+1}}$ are not enabled. We can deduce that this

configuration belongs to $\mathcal{LC}_{m_{i+1}}$, that ends the proof. ■

Lemma 15.10

Starting from any configuration, any execution of $\mathcal{SSMMT}$ under the distributed weakly fair daemon reaches a configuration of $\mathcal{LC}_{IMMT}$ in a finite time.

Proof: Let γ be an arbitrary configuration. We know by Lemma 15.5 that any execution starting from γ reaches in a finite time a configuration of $\mathcal{LC}_{mr} = \mathcal{LC}_{m_0}$. Then, we can apply at most k times the result of Lemma 15.9 to obtain that any execution starting from γ reaches in a finite time a configuration of $\mathcal{LC}_{m_k} = \mathcal{LC}_{IMMT}$, that proves the result. ■

Theorem 15.1

$\mathcal{SSMMT}$ is a $(S_B, n-1)$-TA strictly stabilizing distributed protocol for $spec_{IMMT}$ under the distributed weakly fair daemon.

Proof: This result is a direct consequence of Lemmas 15.4 and 15.10. ■

15.1.3 Proof of Topology-Aware Strong Stabilization

In this section, we prove the $(t, S_B^*, n-1)$-TA strong stabilization of $\mathcal{SSMMT}$ under the distributed k-bounded strongly fair daemon. Note that k may be any arbitrary natural number. Nevertheless, the actual value of k influences the maximal number of disruptions of $\mathcal{SSMMT}$.

In the same way as in the previous section, this proof shares similarities with the corresponding one in Chapter 14. Indeed, the key idea is once again to focus on vertices of $S_B \setminus S_B^*$ after the convergence of $\mathcal{SSMMT}$ on S_B and to prove the two following properties: any such vertex executes a bounded number of steps in any execution and cannot remains unactivated if it does not satisfy $spec_{IMMT}$. The proof of these properties is complexified by the fact that the communication subgraph induced by $S_B \setminus S_B^*$ is not reduced to a set of chains.

Let us denote $E_B = S_B \setminus S_B^*$ (*i.e.* E_B is the set of vertices v such that $max_met(g, v, r) = \max_{b \in B}\{max_met(g, v, b)\}$). Intuitively, E_B gathers the set of vertices that are at equals distance (with respect to $\mathcal{M}$) from the root than the nearest Byzantine vertex. Note that the communication subgraph $g(E_B)$ induced by E_B may have several connected components. In the following, we use the following notations: $E_B = \{E_B^1, \dots, E_B^\ell\}$ where each E_B^i ($i \in \{0, \dots, \ell\}$) is a subset of E_B inducing a maximal (in number of vertices) connected component of g, $g(E_B^i)$ ($i \in \{0, \dots, \ell\}$) is the communication subgraph induced by E_B^i, and then $diam(g(E_B)) = \max_{i \in \{0,\dots,\ell\}}\{diam(g(E_B^i))\}$. When a and b are two integers, we define the following function: $\Pi(a, b) = \frac{a^{b+1}-1}{a-1}$.

Lemma 15.11

If γ is a configuration of $\mathcal{LC}$, then any vertex $v \in E_B$ is activated at most $\Pi(k, diam(g(E_B))) \times deg(g) \times D$ times in any execution starting from γ.

Proof: Let γ be a configuration of $\mathcal{LC}$ and σ be an execution starting from γ. Let p be a vertex of E_B^i ($i \in \{0, \dots, \ell\}$) such that there exists a neighbor q which satisfies $q \in V \setminus S_B$ and $max_met(g, p, r) = met(max_met(g, q, r), w_{p,q})$ (such a vertex exists by construction of E_B^i). We are going to prove by induction on d the following property:

(P_d): if v is a vertex of E_B^i such that $dist(g(E_B^i), p, v) = d$, then v executes at most $\Pi(k, d) \times deg(g) \times D$ actions in σ.

Initialization: $d = 0$.

This implies that $v = p$. Then, by construction, there exists a neighbor q which satisfies $q \in V \setminus S_B$ and $max_met(g, p, r) = met(max_met(g, q, r), w_{p,q})$. As $\gamma \in \mathcal{LC}$, Lemma 15.4 ensures us that $level_q = max_met(g, q, r)$ and $dist_q < D - 1$ in any configuration of σ. Then, the boundedness of $\mathcal{M}$ implies that q belongs to the set which is parameter to the function $next_v$ at any execution of rules **(R_2)** or **(R_3)** by p. Consequently, p executes at most $deg(g)$ times rules **(R_2)** and **(R_3)** in σ before choosing q as its parent. Moreover, note that p can execute rule **(R_1)** at most D times between two consecutive executions of rules **(R_2)** and **(R_3)** (because **(R_1)** only increases $dist_p$ which is bounded by D). Consequently, p executes at most $deg(g) \times D$ actions before choosing q as its parent.

By Lemma 15.4, we know that q takes no action in σ. Once p chooses q as its parent, its state is consistent with the one of q (by construction of rules **(R_2)** and **(R_3)**). Hence, p is never enabled after choosing q as its parent. Consequently, we obtain that p takes at most $deg(g) \times D$ actions in σ, that proves **(P_0)**.

Induction: $d > 0$ and **(P_{d-1})** is true.

Let v be a vertex of E_B^i such that $dist(g(E_B^i), p, v) = d$. By construction, there exists a neighbor u of v which belongs to E_B^i such that $dist(g(E_B^i), p, u) = d-1$. By **$(P_{d-1})$**, we know that u takes at most $\Pi(k, d-1) \times deg(g) \times D$ actions in σ. The k-boundedness of the daemon allows us to conclude that v takes at most $k \times \Pi(k, d-1) \times deg(g) \times D$ actions before the last action of u. Then, a similar reasoning to the one of the initialization part allows us to say that v takes at most $deg(g) \times D$ actions after the last action of u (note that the fact that $|M(S)| \geq 2$, the construction of D and the management of *dist* variables imply that $dist_u < D - 1$ after the last action of u). In conclusion, v takes at most $k \times \Pi(k, d-1) \times deg(g) \times D + deg(g) \times D = \Pi(k, d) \times deg(g) \times D$ actions in σ, that proves **(P_d)**.

As $diam(g(E_B))$ is the maximal diameter of connected components of the communication subgraph induced by E_B, then we know that $dist(g(E_B^i), p, v) \leq diam(g(E_B))$ for any vertex v in E_B^i. For any vertex v of E_B, there exists $i \in \{0, \dots, \ell\}$ such that $v \in E_B^i$. We can deduce that any vertex of E_B takes at most $\Pi(k, diam(g(E_B))) \times deg(g) \times D$ actions in σ, that implies the result. ∎

Lemma 15.12

If γ is a configuration of $\mathcal{LC}$ and v is a vertex such that $v \in E_B$, then for any execution σ starting from γ either

1. *there exists a configuration γ' of σ such that $spec_{IMMT}(v)$ is always satisfied after γ'; or*
2. *v is activated in σ.*

Proof: Let γ be a configuration of $\mathcal{LC}$ and v be a vertex such that $v \in E_B$. By contradiction, assume that there exists an execution σ starting from γ such that (i) $spec_{IMMT}(v)$ is infinitely often false in σ and (ii) v is never activated in σ.

For any configuration γ, let us denote by $P_v(\gamma) = (v_0 = v, v_1 = prnt_v, v_2 = prnt_{v_1}, \ldots, v_k = prnt_{v_{k-1}}, p_v = prnt_{v_k})$ the maximal sequence of vertices following pointers $prnt$ (maximal means here that either $prnt_{p_v} = \bot$ or p_v is the first vertex such that there $p_v = v_i$ for some $i \in \{0, \ldots, k\}$).

Let us study the following cases:

Case 1: $prnt_v \in V \setminus S_B$ in γ.
Since $\gamma \in \mathcal{LC}$, $prnt_v$ satisfies $spec_{IMMT}(prnt_v)$ in γ and in any execution starting from γ (by Lemma 15.4). Hence, $prnt_v$ is never activated in σ. If v does not satisfy $spec_{IMMT}(v)$ in γ, then we have $level_v \neq met(level_{prnt_v}, w_{v,prnt_v})$ or $dist_v \neq 0$ in γ. Then, v is continuously enabled in σ and we have a contradiction between assumption (ii) and the strong fairness of the daemon. This implies that v satisfies $spec_{IMMT}(v)$ in γ. The fact that $prnt_v$ is never activated in γ and that the state of v is consistent with the one of $prnt_v$ ensures us that v is never enabled in any execution starting from γ. Hence, $spec_{IMMT}(v)$ remains true in any execution starting from γ. This contradicts the assumption (i) on σ.

Case 2: $prnt_v \notin V \setminus S_B$ in γ.
By the assumption (i) on σ, we can deduce that there exists infinitely many configurations γ' such that a vertex of $P_v(\gamma')$ is enabled (since $spec_{IMMT}(v)$ is false only when the state of a vertex of $P_v(\gamma')$ is not consistent with the one of its parent that made it enabled). By construction, the length of $P_v(\gamma')$ is finite for any configuration γ' and there exists only a finite number of vertices in the communication graph. Consequently, there exists at least one vertex which is infinitely often enabled in σ. Since the daemon is strongly fair, we can conclude that there exists at least one vertex which is infinitely often activated in σ.

Let A_σ be the set of vertices which are infinitely often activated in σ. Note that $v \notin A_\sigma$ by assumption (ii) on σ. Let σ' be the suffix of σ starting from γ' which contains only activations of vertices of A_σ. Let p be the first vertex of $P_v(\gamma')$ which belongs to A_σ (p exists since at least one vertex of P_v is enabled when $spec_{IMMT}(v)$ is false). By construction, the prefix of $P_v(\gamma'')$ from v to p in any configuration γ'' of σ remains the same as the one of $P_v(\gamma')$. Let p' be the vertex such that $prnt_{p'} = p$ in σ' (p' exists since $v \neq p$ implies that the prefix of $P_v(\gamma')$ from v to p counts at least two vertices). As p is infinitely often activated and as any activation of p modifies the value of $level_p$ or of $dist_p$ (at least one of these two variables takes at least two different values in σ'), we can deduce that p' is infinitely often enabled in σ' (since the value of

$level_{p'}$ is constant by construction of σ' and p). Since the daemon is strongly fair, p' is activated in a finite time in σ', that contradicts the construction of p.

In the two cases, we obtain a contradiction with the construction of σ, that proves the result. ■

Let $\mathcal{LC}^*_{IMMT}$ be the following set of configurations:

$$\mathcal{LC}^*_{IMMT} = \left\{\gamma \in \Gamma \middle| (\gamma \textit{ is } S^*_B\textit{-legitimate for } spec_{IMMT}) \wedge (IM_{m_k}(\gamma) = true)\right\}$$

Note that, as $S^*_B \subseteq S_B$, we can deduce that $\mathcal{LC}^*_{IMMT} \subseteq \mathcal{LC}_{IMMT}$. Hence, properties of Lemmas 15.11 and 15.12 also apply to configurations of $\mathcal{LC}^*_{IMMT}$.

Lemma 15.13

Any configuration of $\mathcal{LC}^*_{IMMT}$ is $(n \times \Pi(k, diam(g(E_B)) \times deg(g) \times D, \Pi(k, diam(g(E_B)) \times deg(g) \times D, S^*_B, n-1)$-TA time contained for $spec_{IMMT}$.

Proof: Let γ be a configuration of $\mathcal{LC}^*_{IMMT}$. As $S^*_B \subseteq S_B$, we know by Lemma 15.4 that any vertex v of $V \setminus S_B$ satisfies $spec_{IMMT}(v)$ and takes no action in any execution starting from γ.

Let v be a vertex of E_B. By Lemmas 15.11 and 15.12, we know that v takes at most $\Pi(k, diam(g(E_B)) \times deg(g) \times D$ actions in any execution starting from γ. Moreover, we know that v satisfies $spec_{IMMT}(v)$ after its last action (otherwise, we obtain a contradiction between the two lemmas). Hence, any vertex of E_B takes at most $\Pi(k, diam(g(E_B)) \times deg(g) \times D$ actions and then, there are at most $n \times \Pi(k, diam(g(E_B)) \times deg(g) \times D$ S^*_B-TA disruptions in any execution starting from γ (since $|E_B| \leq n$).

By definition of a TA time contained configuration, we obtain the result. ■

Lemma 15.14

Starting from any configuration, any execution of $\mathcal{SSMMT}$ under the distributed k-bounded strongly fair daemon reaches a configuration of $\mathcal{LC}^*_{IMMT}$ in a finite time.

Proof: Let γ be an arbitrary configuration. We know by Lemma 15.10 that any execution starting from γ reaches in a finite time a configuration γ' of $\mathcal{LC}_{IMMT}$.

Let v be a vertex of E_B. By Lemmas 15.11 and 15.12, we know that v takes at most $\Pi(k, diam(g(E_B)) \times deg(g) \times D$ actions in any execution starting from γ'. Moreover, we know that v satisfies $spec_{IMMT}(v)$ after its last action (otherwise, we obtain a contradiction between the two lemmas). This implies that any execution starting from γ' reaches a configuration γ'' such that any vertex v of E_B satisfies $spec_{IMMT}(v)$. It is easy to see that $\gamma'' \in \mathcal{LC}^*_{IMMT}$, that ends the proof. ■

Theorem 15.2

$\mathcal{SSMMT}$ is a $(n \times \Pi(k, diam(g(E_B)) \times deg(g) \times D, S^*_B, n-1)$-TA strongly

stabilizing distributed protocol for $spec_{IMMT}$ under the distributed k-bounded strongly fair daemon.

Proof: This result is a direct consequence of Lemmas 15.13 and 15.14. ■

15.2 Optimality of Containment Areas

This section presents two impossibility results that prove the optimality of containment areas provided by the distributed protocol of the previous section. Indeed, Theorem 15.3 states that there exists no topology-aware strictly stabilizing distributed protocol for maximum metric spanning tree construction for any containment area strictly included in S_B while Theorem 15.4 proves that there exists no topology-aware strongly stabilizing distributed protocol for maximum metric spanning tree construction for any containment area strictly included in S_B^*.

These proofs are based on the construction of a communication graph (depending of the characteristic of the considered maximizable metric) and of a Byzantine behavior that allows us to invalidate the topology-aware strict (respectively strong) stabilization of any distributed protocol exhibiting better containment areas than $\mathcal{SSMMT}$. Note that the Byzantine behavior is simply an alternate root and correct behavior.

15.2.1 Topology-Aware Strict Stabilization

Theorem 15.3

Given a maximizable metric $\mathcal{M} = (M, W, mr, met, \prec)$, even under the central daemon, there exists no $(A_B, 1)$-TA strictly stabilizing distributed protocol for $spec_{IMMT}$ with respect to $\mathcal{M}$ where $A_B \subsetneq S_B$.

Proof: Let $\mathcal{M} = (M, W, mr, met, \prec)$ be a maximizable metric and π be a $(A_B, 1)$-TA strictly stabilizing distributed protocol for $spec_{iMMT}$ with respect to $\mathcal{M}$ where $A_B \subsetneq S_B$. We must distinguish the following cases:

Case 1: $|M| = 1$.

Denote by m the metric value such that $M = \{m\}$. For any communication graph and for any vertex $v \neq r$, we have:

$$max_met(g, v, r) = \min_{b \in B}{}_{\prec}\{max_met(g, v, b)\} = m$$

Consequently, $S_B = V \setminus (B \cup \{r\})$ for any communication graph.

Consider the following communication graph: $V = \{r, u, v, b\}$ and $E = \{\{r, u\}, \{u, v\}, \{v, b\}\}$ (b is a Byzantine vertex). As $S_B = \{u, v\}$ and $A_B \subsetneq S_B$, we have: $u \notin A_B$ or $v \notin A_B$. Consider now the following configuration γ_0^0: $prnt_r = prnt_b = \perp$, $prnt_v = b$, $prnt_u = v$, $level_r = level_u = level_v = level_b = m$, $dist_r = dist_b = 0$, $dist_v = 1$ and $dist_u = 2$ (see Figure 15.4, other variables may have arbitrary values). Note that γ_0^0 is A_B-legitimate for $spec_{IMMT}$ (whatever A_B is).

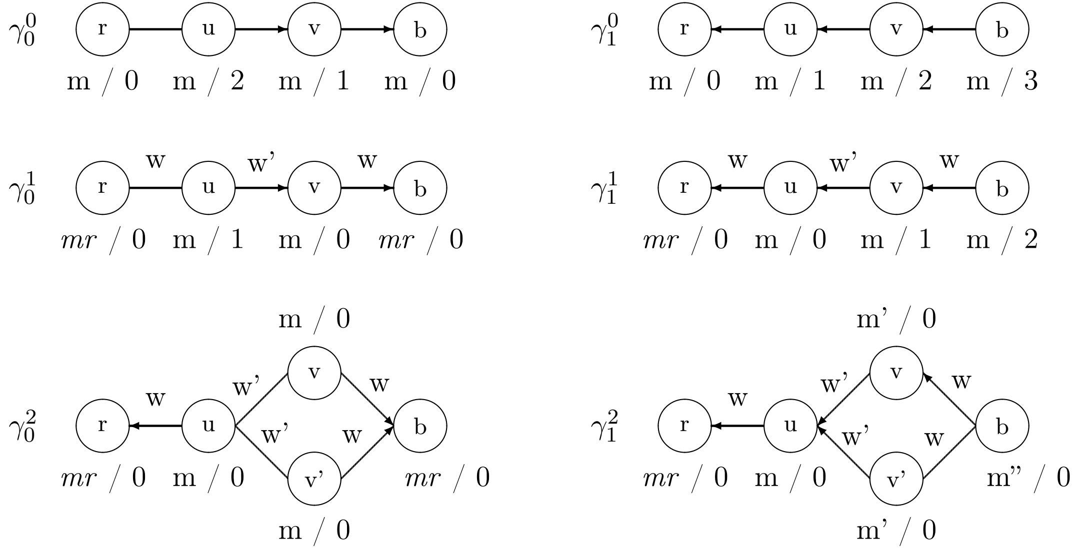

Figure 15.4: Configurations used in proof of Theorem 15.3.

Assume now that b behaves as a correct vertex with respect to π. Then, by convergence of π in a fault-free system starting from γ_0^0 which is not $\emptyset$-legitimate (remember that a topology-aware strictly stabilizing distributed protocol is a special case of self-stabilizing distributed protocol), we can deduce that π reaches in a finite time a configuration γ_1^0 (see Figure 15.4) in which: $prnt_r = \perp$, $prnt_u = r$, $prnt_v = u$, $prnt_b = v$, $level_r = level_u = level_v = level_b = m$, $dist_r = 0$, $dist_u = 1$, $dist_v = 2$ and $dist_b = 3$. Note that vertices u and v modify their O-variables in this execution. This contradicts the $(A_B, 1)$-TA strict stabilization of π (whatever A_B is).

Case 2: $|M| \geq 2$.

By definition of a bounded metric, we can deduce that there exist $m \in M$ and $w \in W$ such that $m = met(mr, w) \prec mr$. Then, we must distinguish the following cases:

Case 2.1: m is a fixed point of $\mathcal{M}$.

Consider the following communication graph: $V = \{r, u, v, b\}$, $E = \{\{r, u\}, \{u, v\}, \{v, b\}\}$, $w_{r,u} = w_{v,b} = w$, and $w_{u,v} = w'$ (b is a Byzantine vertex). As for any $w' \in W$, $met(m, w') = m$ (by definition of a fixed point), we have: $S_B = \{u, v\}$. Since $A_B \subsetneq S_B$, we have: $u \notin A_B$ or $v \notin A_B$. Consider now the following configuration γ_0^1: $prnt_r = prnt_b = \perp$, $prnt_v = b$, $prnt_u = v$, $level_r = level_b = mr$, $level_u = level_v = m$, $dist_r = dist_b = 0$, $dist_v = 0$ and $dist_u = 1$ (see Figure 15.4, other variables may have arbitrary values). Note that γ_0^1 is A_B-legitimate for $spec_{IMMT}$ (whatever A_B is).

Assume now that b behaves as a correct vertex with respect to π. Then, by convergence of π in a fault-free system starting from γ_0^1 which is not $\emptyset$-legitimate (remember that a topology-aware strictly stabilizing distributed protocol is a special case of self-stabilizing distributed protocol), we can deduce that π reaches in a finite time a configuration γ_1^1 (see Figure 15.4) in which: $prnt_r = \perp$, $prnt_u = r$, $prnt_v = u$, $prnt_b = v$, $level_r = mr$, $level_u = level_v = level_b = m$ (since m is a fixed point),

$dist_r = 0$, $dist_u = 0$, $dist_v = 1$ and $dist_b = 2$. Note that vertices u and v modify their O-variables in this execution. This contradicts the $(A_B, 1)$-TA strict stabilization of π (whatever A_B is).

Case 2.2: m is not a fixed point of $\mathcal{M}$.

This implies that there exists $w' \in W$ such that: $met(m, w') \prec m$ (remember that $\mathcal{M}$ is bounded). Consider the following communication graph: $V = \{r, u, v, v', b\}$, $E = \{\{r, u\}, \{u, v\}, \{u, v'\}, \{v, b\}, \{v', b\}\}$, $w_{r,u} = w_{v,b} = w_{v',b} = w$, and $w_{u,v} = w_{u,v'} = w'$ (b is a Byzantine vertex). We can see that $S_B = \{v, v'\}$. Since $A_B \subsetneq S_B$, we have: $v \notin A_B$ or $v' \notin A_B$. Consider now the following configuration γ_0^2: $prnt_r = prnt_b = \perp$, $prnt_v = prnt_{v'} = b$, $prnt_u = r$, $level_r = level_b = mr$, $level_u = level_v = level_{v'} = m$, $dist_r = dist_b = 0$, $dist_v = dist_{v'} = 0$ and $dist_u = 0$ (see Figure 15.4, other variables may have arbitrary values). Note that γ_0^2 is A_B-legitimate for $spec_{IMMT}$ (whatever A_B is).

Assume now that b behaves as a correct vertex with respect to π. Then, by convergence of π in a fault-free system starting from γ_0^2 which is not $\emptyset$-legitimate (remember that a topology-aware strictly stabilizing distributed protocol is a special case of self-stabilizing distributed protocol), we can deduce that π reaches in a finite time a configuration γ_1^2 (see Figure 15.4) in which: $prnt_r = \perp$, $prnt_u = r$, $prnt_v = prnt_{v'} = u$, $prnt_b = v$ (or $prnt_b = v'$), $level_r = mr$, $level_u = m$ $level_v = level_{v'} = met(m, w') = m'$, $level_b = met(m', w) = m''$, $dist_r = 0$, $dist_u = 0$, $dist_v = dist_{v'} = 0$ and $dist_b = 0$. Note that vertices v and v' modify their O-variables in this execution. This contradicts the $(A_B, 1)$-TA strict stabilization of π (whatever A_B is). ∎

15.2.2 Topology-Aware Strong Stabilization

Theorem 15.4

Given a maximizable metric $\mathcal{M} = (M, W, mr, met, \prec)$, even under the central daemon, there exists no $(t, A_B^*, 1)$-TA strongly stabilizing distributed protocol for $spec_{IMMT}$ with respect to $\mathcal{M}$ where $A_B^* \subsetneq S_B^*$ and t is a given finite integer.

Proof: Let $\mathcal{M} = (M, W, mr, met, \prec)$ be a maximizable metric and π be a $(t, A_B^*, 1)$-TA strongly stabilizing protocol for $spec_{IMMT}$ with respect to $\mathcal{M}$ where $A_B^* \subsetneq S_B^*$ and t is a finite integer. We must distinguish the following cases:

Case 1: $|M| = 1$.

Denote by m the metric value such that $M = \{m\}$. For any communication graph and for any vertex v, we have:

$$max_met(g, v, r) = \min_{\prec, b \in B}\{max_met(g, v, b)\} = m$$

Consequently, $S_B^* = \emptyset$ for any communication graph. Then, it is absurd to have $A_B^* \subsetneq S_B^*$.

Case 2: $|M| \geq 2$.

By definition of a bounded metric, we can deduce that there exists $m \in M$

and $w \in W$ such that $m = met(mr, w) \prec mr$. Then, we must distinguish the following cases:

Case 2.1: m is a fixed point of $\mathcal{M}$.

Let g be a communication graph such that any edge incident to the root or a Byzantine vertex has a weight equals to w. Then, we can deduce that we have:

$$\begin{cases} m = \max_{\prec, b \in B}\{max_met(g, r, b)\} \prec max_met(g, r, r) = mr \\ \forall v \in V \setminus (B \cup \{r\}), max_met(g, v, r) = \max_{\prec, b \in B}\{max_met(g, v, b)\} = m \end{cases}$$

Hence, $S_B^* = \emptyset$ for any such communication graph. Then, it is absurd to have $A_B^* \subsetneq S_B^*$.

Case 2.2: m is not a fixed point of $\mathcal{M}$.

This implies that there exists $w' \in W$ such that: $met(m, w') \prec m$ (remember that $\mathcal{M}$ is bounded). Consider the following communication graph: $V = \{r, u, u', v, v', b\}$, $E = \{\{r, u\}, \{r, u'\}, \{u, v\}, \{u', v'\}, \{v, b\}, \{v', b\}\}$, $w_{r,u} = w_{r,u'} = w_{v,b} = w_{v',b} = w$, and $w_{u,v} = w_{u',v'} = w'$ (b is a Byzantine vertex). We can see that $S_B^* = \{v, v'\}$. Since $A_B^* \subsetneq S_B$, we have: $v \notin A_B^*$ or $v' \notin A_B^*$. Consider now the following configuration γ_0: $prnt_r = prnt_b = \bot$, $level_r = level_b = mr$, $dist_r = dist_b = 0$ and $prnt$, $level$, and $dist$ variables of other vertices are arbitrary (see Figure 15.5, other variables may have arbitrary values but other variables of b are identical to those of r).

Assume now that b executes exactly the same actions as r (if any) immediately after r (note that $r \notin A_B^*$ and hence $prnt_r = \bot$, $level_r = mr$, and $dist_r = 0$ still hold by closure and then $prnt_b = \bot$, $level_b = mr$, and $dist_r = 0$ still hold too). Then, by symmetry of the execution and by convergence of π to $spec_{IMMT}$, we can deduce that π reaches in a finite time a configuration γ_1 (see Figure 15.5) in which: $prnt_r = prnt_b = \bot$, $prnt_u = prnt_{u'} = r$, $prnt_v = prnt_{v'} = b$, $level_r = level_b = mr$, $level_u = level_{u'} = level_v = level_{v'} = m$, and $\forall v \in V, dist_v = legal_dist_{prnt_v}$ (because this configuration is the only one in which all correct vertex v satisfies $spec_{IMMT}(v)$ when $prnt_r = prnt_b = \bot$ and $level_r = level_b = mr$ since $met(m, w') \prec m$). Note that γ_1 is A_B^*-legitimate for $spec_{IMMT}$ and A_B^*-stable (whatever A_B^* is).

Assume now that b behaves as a correct vertex with respect to π. Then, by convergence of π in a fault-free system starting from γ_1 which is not $\emptyset$-legitimate (remember that a TA strongly stabilizing distributed protocol is a special case of self-stabilizing distributed protocol), we can deduce that π reaches in a finite time a configuration γ_2 (see Figure 15.5) in which: $prnt_r = \bot$, $prnt_u = prnt_{u'} = r$, $prnt_v = u$, $prnt_{v'} = u'$, $prnt_b = v$ (or $prnt_b = v'$), $level_r = mr$, $level_u = level_{u'} = m$ $level_v = level_{v'} = met(m, w') = m'$, $level_b = met(m', w) = m''$, and $\forall v \in V, dist_v = legal_dist_{prnt_v}$. Note that vertices v and v' modify their O-variables in the portion of execution between γ_1 and γ_2 and that γ_2 is A_B^*-legitimate for $spec_{IMMT}$ and A_B^*-stable (whatever A_B^* is). Consequently, this portion of execution contains at least one A_B^*-TA disruption (whatever A_B^* is).

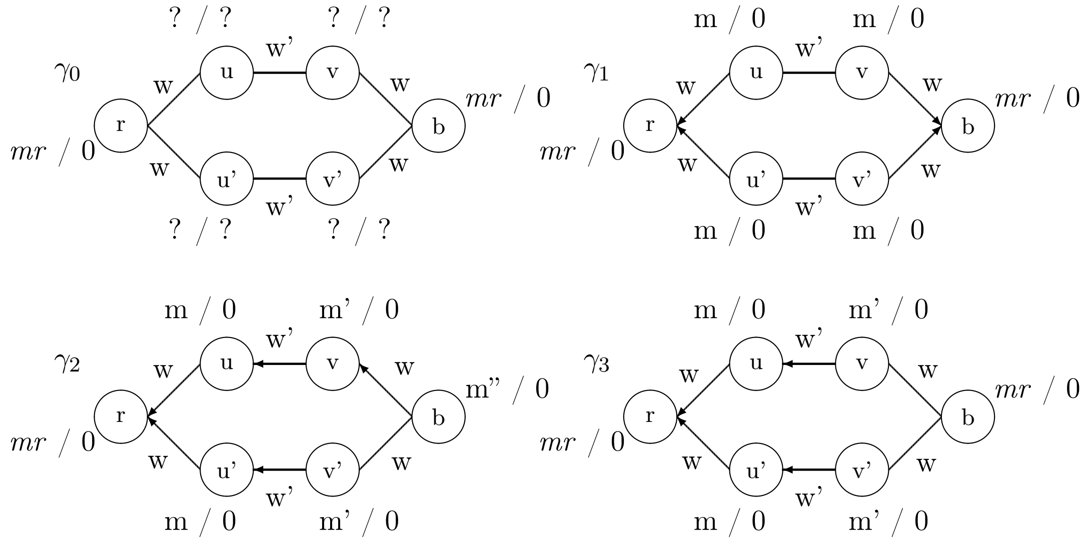

Figure 15.5: Configurations used in proof of Theorem 15.4.

Assume now that the Byzantine vertex b takes the following state: $prnt_b = \perp$, $level_b = mr$, and $dist_b = 0$. This action brings the system into configuration γ_3 (see Figure 15.5). From this configuration, we can repeat the execution we constructed from γ_0. By the same token, we obtain an execution of π which contains A_B^*-legitimate and A_B^*-stable configurations (see γ_1) and an infinite number of A_B^*-TA disruptions (whatever A_B^* is) which contradicts the $(t, A_B^*, 1)$-TA strong stabilization of π. ■

15.3 Strong Stabilization

In this section, we discuss about the relationship between topology-aware strong stabilization and strong stabilization on maximum metric spanning tree construction. We characterize by a necessary and sufficient condition the set of assigned metric that allow strong stabilization. Indeed, properties on the metric itself are not sufficient to conclude on the possibility of strong stabilization: we must know information about the considered communication graph (assignation of the metric). Informally, it is possible to construct a maximum metric spanning tree in a strongly stabilizing way if and only if the considered metric is strongly maximizable (that is, if the metric is strictly decreasing and has one and only one fixed point, see Definition 15.5) and if the desired containment radius is sufficiently large with respect to the size of the set of used metric values of the communication graph.

First, we define a specific class of maximizable metrics (strongly maximizable metrics, see Definition 15.5). Then, we prove an impossibility result that state that it is impossible to construct a maximum metric spanning tree in a strongly stabilizing way if we do not consider such a metric (see Lemma 15.15). Finally, we provide our full characterization of maximizable metrics that allow strong stabilization (see

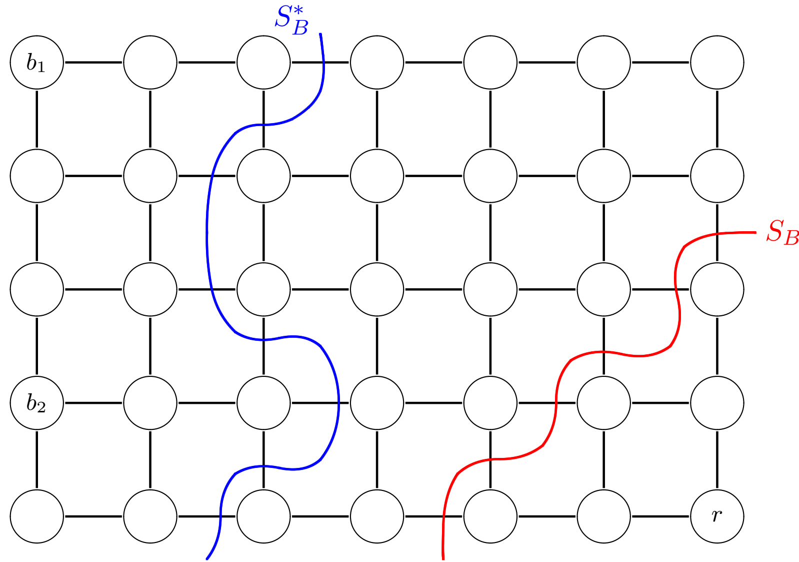

Figure 15.6: Example of containment areas for *MET*.

Theorem 15.5).

Definition 15.5 (*Strongly maximizable metric*)

A maximizable metric $\mathcal{M} = (M, W, mr, met, \prec)$ is strongly maximizable if and only if $|M| = 1$ or if the following properties holds:

- *$|M| \geq 2$;*
- *$\mathcal{M}$ is strictly decreasing; and*
- *$\mathcal{M}$ has one and only one fixed point.*

Note that $\mathcal{NC}$ is a strongly maximizable metric (since $|M_4| = 1$) whereas $\mathcal{BFS}$ or $\mathcal{SP}$ are not (since the first one has no fixed point, the second is not strictly decreasing). If we consider the metric *MET* defined below, we can show that *MET* is a strongly maximizable metric such that $|M| \geq 2$.

$$\begin{array}{rcl} MET & = & (M_5, W_5, met_5, mr_5, \prec_5) \\ where & & M_5 = \{0,1,2,3\} \\ & & W_5 = \{1\} \\ & & met_5(m,w) = max\{0, m-w\} \\ & & mr_5 = 3 \\ & & \prec_5 \ \ is\ the\ classical\ < \ relation \end{array}$$

Figure 15.6 provides an illustration of containment areas S_B and S_B^* for this metric. Note that S_B^* is equals to the union of the 2-neighborhood of Byzantine vertices.

Using this definition, we can now provide an impossibility result that generalizes Theorem 14.2 (impossibility of strong stabilization for $spec_{BFST}$). Note that the proof relies also on similar ideas while we must generalize it to consider any maximizable metric.

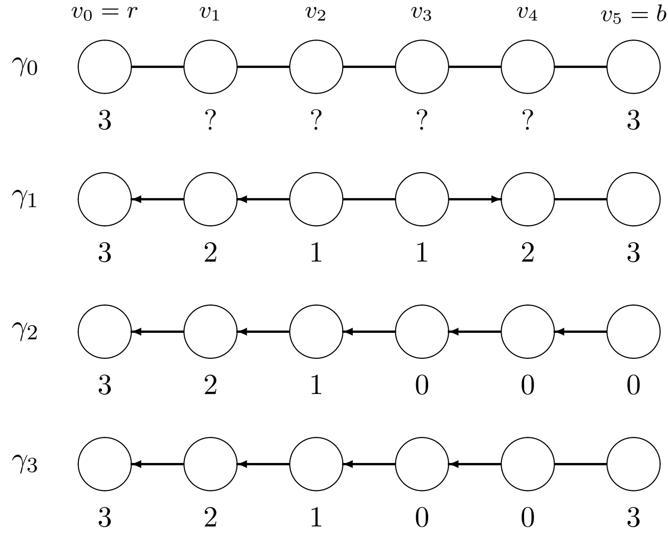

Figure 15.7: Illustration of configurations used in proof of Lemma 15.15, case 1 for the metric MET with $c = 1$.

Lemma 15.15

Given a maximizable metric $\mathcal{M} = (M, W, mr, met, \prec)$, even under the central daemon, there exists no $(t, c, 1)$-strongly stabilizing distributed protocol for $spec_{IMMT}$ with respect to $\mathcal{M}$ for any finite integer t if:

$$\begin{cases} \mathcal{M} \textit{ is not a strongly maximizable metric, or} \\ c < |M| - 2 \end{cases}$$

Proof: We prove this result by contradiction. We assume that $\mathcal{M} = (M, W, mr, met, \prec)$ is a maximizable metric such that there exist a finite integer t and a distributed protocol π that is a $(t, c, 1)$-strongly stabilizing distributed protocol for $spec_{IMMT}$ with respect to $\mathcal{M}$. We distinguish the following cases (note that they are exhaustive):

Case 1: $\mathcal{M}$ is a strongly maximizing metric and $c < |M| - 2$.

As $c \geq 0$, we know that $|M| \geq 2$ and then, by definition of a strongly stabilizing metric, $\mathcal{M}$ is strictly decreasing and has one and only one fixed point.

By assumption on $\mathcal{M}$, we know that there exist $c+3$ distinct metric values $m_0 = mr, m_1, \ldots, m_{c+2}$ in M and $w_0, w_1, \ldots, w_{c+1}$ in W such that: $\forall i \in \{1, \ldots, c+2\}, m_i = met(m_{i-1}, w_{i-1}) \prec m_{i-1}$.

Let g be the following communication graph $V = \{v_0 = r, v_1, \ldots, v_{2c+2}, v_{2c+3} = b\}$, $E = \{\{v_i, v_{i+1}\}, i \in \{0, \ldots, 2c+2\}\}$ and $\forall i \in \{0, c+1\}, w_{v_i, v_{i+1}} = w_{v_{2c+3-i}, v_{2c+2-i}} = w_i$. Note that the choice $w_{v_{c+1}, v_{c+2}} = w_{c+1}$ ensures us the following property when $level_r = level_b = mr$:

$$\begin{cases} max_met(g, v_{c+1}, b) \prec max_met(g, v_{c+1}, r) \\ max_met(g, v_{c+2}, r) \prec max_met(g, v_{c+2}, b) \end{cases}$$

Vertex v_0 is the real root and vertex b is a Byzantine one. Note that the construction of g ensures the following properties when $level_r = level_b = mr$:

$$\begin{cases} \forall i \in \{1, \dots, c+1\}, max_met(g, v_i, r) = max_met(g, v_{2c+3-i}, b) \\ max_met(g, v_i, b) \prec max_met(g, v_i, r) \\ max_met(g, v_{2c+3-i}, r) \prec max_met(g, v_{2c+3-i}, b) \end{cases}$$

Assume that the initial configuration γ_0 of g satisfies: $prnt_r = prnt_b = \perp$, $level_r = level_b = mr$, and other variables of b (in particular $dist$) are identical to those of r (see Figure 15.7, variables of other vertices may be arbitrary). Assume now that b takes exactly the same actions as r (if any) immediately after r. Then, by symmetry of the execution and by convergence of π to $spec_{IMMT}$, we can deduce that π reaches in a finite time a configuration γ_1 (see Figure 15.7) in which: $\forall i \in \{1, \dots, c+1\}, prnt_{v_i} = v_{i-1}$, $level_{v_i} = max_met(g, v_i, r) = m_i$, $dist_{v_i} = legal_dist(v_i, prnt_{v_i})$ and $\forall i \in \{c+2, \dots, 2c+2\}, prnt_{v_i} = v_{i+1}$, $level_{v_i} = max_met(g, v_i, b) = m_{2c+3-i}$, and $dist_{v_i} = legal_dist(v_i, prnt_{v_i})$ (because this configuration is the only one in which all correct vertices v satisfy $spec_{IMMT}(v)$ when $prnt_r = prnt_b = \perp$ and $level_r = level_b = mr$ by construction of g). Note that γ_1 is c-legitimate and c-stable.

Assume now that the Byzantine vertex acts as a correct vertex and executes correctly π. Then, by convergence of π in fault-free systems (remember that a strongly-stabilizing distributed protocol is a special case of self-stabilizing distributed protocol), we can deduce that π reaches in a finite time a configuration γ_2 (see Figure 15.7) in which: $\forall i \in \{1, \dots, 2c+3\}, prnt_{v_i} = v_{i-1}$, $level_{v_i} = max_met(g, v_i, r)$, and $dist_{v_i} = legal_dist(v_i, prnt_{v_i})$ (because this configuration is the only one in which all vertices v satisfy $spec_{IMMT}(v)$). Note that the portion of execution between γ_1 and γ_2 contains at least one c-disruption (v_{c+2} is a c-correct vertex and modifies at least once its O-variables) and that γ_2 is c-legitimate and c-stable.

Assume now that the Byzantine vertex b takes the following state: $prnt_b = \perp$ and $level_b = mr$. This action brings the system into configuration γ_3 (see Figure 15.7). From this configuration, we can repeat the execution we constructed from γ_0. By the same token, we obtain an infinite execution of π that contains c-legitimate and c-stable configurations (see γ_1) and an infinite number of c-disruptions that contradicts the $(t, c, 1)$-strong stabilization of π.

Case 2: $\mathcal{M}$ is not strictly decreasing.

By definition, we know that $\mathcal{M}$ is not a strongly maximizable metric. Hence, we have $|M| \geq 2$. Then, the definition of a strictly decreasing metric implies that there exists a metric value $m \in M$ such that: $\exists w \in W$, $met(m, w) = m$ and $\exists w' \in W, m' = met(m, w') \prec m$ (and thus m is not a fixed point of $\mathcal{M}$). By the utility condition on M, we know that there exists a sequence of metric values $m_0 = mr, m_1, \dots, m_l = m$ in M and $w_0, w_1, \dots, w_{l-1}$ in W such that $\forall i \in \{1, \dots, l\}, m_i = met(m_{i-1}, w_{i-1})$. Denote by k the length of the shortest such sequence. Note that this implies that $\forall i \in \{1, \dots, k\}, m_i \prec m_{i-1}$ (otherwise we can remove m_i from the sequence and this is contradictory with the construction of k). We distinguish the following cases:

Case 2.1: $k \geq c+2$.

We can use the same token as case 1 above by using w' instead of w_{c+1} in the case where $k = c+2$ (since we know that $met(m, w') \prec m$).

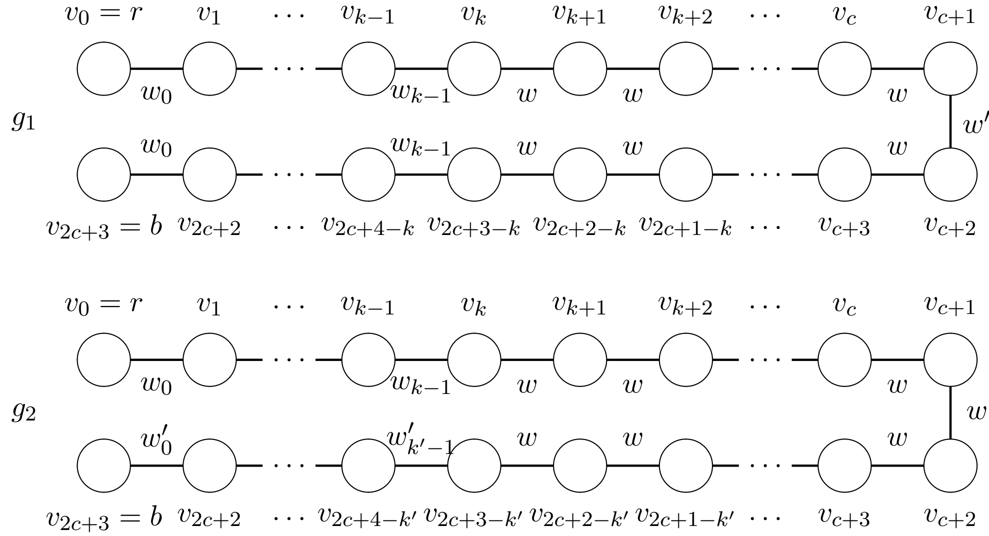

Figure 15.8: Configurations used in proof of Lemma 15.15, cases 2 and 3.

Case 2.2: $k < c + 2$.

Let g_1 be the following communication graph $V = \{v_0 = r, v_1, \ldots, v_{2c+2}, v_{2c+3} = b\}$, $E = \{\{v_i, v_{i+1}\}, i \in \{0, \ldots, 2c+2\}\}$, and the following weight function (see Figure 15.8):

$$\begin{cases} \forall i \in \{0, \ldots, k-1\}, w_{v_i, v_{i+1}} = w_{v_{2c+3-i}, v_{2c+2-i}} = w_i \\ \forall i \in \{k, \ldots, c\}, w_{v_i, v_{i+1}} = w_{v_{2c+3-i}, v_{2c+2-i}} = w \\ w_{v_{c+1}, v_{c+2}} = w' \end{cases}$$

Note that this choice ensures us the following property when $level_r = level_b = mr$:

$$\begin{cases} max_met(g, v_{c+1}, b) \prec max_met(g, v_{c+1}, r) \\ max_met(g, v_{c+2}, r) \prec max_met(g, v_{c+2}, b) \end{cases}$$

Vertex v_0 is the real root and vertex b is a Byzantine one. Note that the construction of g ensures the following properties when $level_r = level_b = mr$:

$$\begin{cases} \forall i \in \{1, \ldots, c+1\}, max_met(g, v_i, r) = max_met(g, v_{2c+3-i}, b) \\ max_met(g, v_i, b) \prec max_met(g, v_i, r) \\ max_met(g, v_{2c+3-i}, r) \prec max_met(g, v_{2c+3-i}, b) \end{cases}$$

This construction allows us to follow the same proof as in case 1 above.

Case 3: $\mathcal{M}$ has no or more than two fixed point, and is strictly decreasing.

If $\mathcal{M}$ has no fixed point and is strictly decreasing, then $|M|$ is not finite and then, we can apply the result of case 1 above since c is a finite integer.

If $\mathcal{M}$ has two or more fixed points and is strictly decreasing, denote by Υ and Υ' two fixed points of $\mathcal{M}$. Without loss of generality, assume that $\Upsilon \prec \Upsilon'$. By the utility condition on M, we know that there exists sequences of metric values $m_0 = mr, m_1, \ldots, m_l = \Upsilon$ and $m_0' = mr, m_1', \ldots, m_{l'}' = \Upsilon'$ in M and

$w_0, w_1, \dots, w_{l-1}$ and $w'_0, w'_1, \dots, w'_{l'-1}$ in W such that $\forall i \in \{1, \dots, l\}, m_i = met(m_{i-1}, w_{i-1})$ and $\forall i \in \{1, \dots, l'\}, m'_i = met(m'_{i-1}, w'_{i-1})$. Denote by k and k' the length of shortest such sequences. Note that this implies that $\forall i \in \{1, \dots, k\}, m_i \prec m_{i-1}$ and $\forall i \in \{1, \dots, k'\}, m'_i \prec m'_{i-1}$ (otherwise we can remove m_i or m'_i from the corresponding sequence). We distinguish the following cases:

Case 3.1: $k > c+2$ or $k' > c+2$.

Without loss of generality, assume that $k > c+2$ (the second case is similar). We can use the same token as case 1 above.

Case 3.2: $k \leq c+2$ and $k' \leq c+2$.

Let w be an arbitrary value of W. Let g_2 be the following communication graph $V = \{v_0 = r, v_1, \dots, v_{2c+2}, v_{2c+3} = b\}$, $E = \{\{v_i, v_{i+1}\}, i \in \{0, \dots, 2c+2\}\}$, and the following weight function (see Figure 15.8):

$$\begin{cases} \forall i \in \{0, k-1\}, w_{v_i, v_{i+1}} = w_i \\ \forall i \in \{0, k'-1\}, w_{v_{2c+3-i}, v_{2c+2-i}} = w'_i \\ \forall i \in \{k, 2c+2-k'\}, w_{v_i, v_{i+1}} = w \end{cases}$$

Note that this choice ensures us the following property when $level_r = level_b = mr$:

$$\begin{cases} max_met(g, v_{c+1}, r) = \Upsilon \prec \Upsilon' = max_met(g, v_{c+1}, b) \\ max_met(g, v_{c+2}, r) = \Upsilon \prec \Upsilon' = max_met(g, v_{c+2}, b) \end{cases}$$

Vertex v_0 is the real root and vertex b is a Byzantine one. This construction allows us to follow a similar proof as in case 1 above (note that any vertex v_i which satisfies $max_met(g, v_i, r) \prec \Upsilon'$ will be disturbed infinitely often, in particular at least v_{c+1} and v_{c+2} which contradicts the $(t, c, 1)$-strong stabilization of π).

In any case, we show that there exists a communication graph that contradicts the $(t, c, 1)$-strong stabilization of π that ends the proof. ■

Using this impossibility result, we can now provide our necessary and sufficient condition for strong stabilization for maximum metric spanning tree construction.

Theorem 15.5

Given an assigned metric $\mathcal{AM} = (M, W, mr, met, \prec, wf)$ over a communication graph g, there exists a $(t, c, n-1)$-strongly stabilizing protocol for $spec_{IMMT}$ with a finite t if and only if:

$$\begin{cases} (M, W, met, mr, \prec) \text{ is a strongly maximizable metric, and} \\ c \geq max\{0, |M(g)| - 2\} \end{cases}$$

Proof: We split this proof into two parts:

1) Proof of the "if" part: Denote $(M, W, met, mr, \prec)$ by $\mathcal{M}$ and assume that $\mathcal{M}$ is a strongly maximizable metric and that $c \geq max\{0, |M(g)| - 2\}$. We distinguish the following cases:

	Containment	Result	Proved by...
(c, f)-strict stabilization	$c \in \mathbb{N}, f = 1$	Impossible	[NA02]
(t, c, f)-strong stabilization	$c \in \mathbb{N}, t \in \mathbb{N}, f = n - 1$	Possible $\Leftrightarrow \mathcal{C}(\mathcal{M}, c)$	Theorem 15.5
(C_B, f)-TA strict stabilization	$C_B \subsetneq S_B, f = 1$	Impossible	Theorem 15.3
	$C_B = S_B, f = n - 1$	Possible	Theorem 15.1
(t, C_B, f)-TA strong stabilization	$C_B \subsetneq S_B^*, f = 1$	Impossible	Theorem 15.4
	$C_B = S_B^*, f = n - 1$	Possible	Theorem 15.2

Table 15.1: Summary of results of Chapter 15 related to $spec_{IMMT}$ with $\mathcal{C}(\mathcal{M}, c)$ a predicate that is true if and only if $\mathcal{M} = (M, W, mr, met, \prec)$ is a strongly maximizable metric and $c \geq max\{0, |M(g)| - 2\}$.

Case 1: $|M(g)| = 1$ (and hence $c \geq 0$).
Denote by m the metric value such that $M(g) = \{m\}$. For any correct vertex v, we have $max_met(g, v, r) = \underset{b \in B}{min_{\prec}}\{max_met(g, v, b)\} = m$. We can deduce that it is equivalent to construct a maximum metric spanning tree for $\mathcal{M}$ and for $\mathcal{NC}$ over any communication graph. By Theorem 14.1, we know that there exists a $(t, 0, n-1)$-strongly stabilizing protocol for this problem with a finite t, that proves the result.

Case 2: $|M(g)| \geq 2$ (and hence $c \geq |M(g)| - 2$).
By Theorem 15.2, we know that there exists a $(t, S_B^*, n-1)$-TA strongly stabilizing protocol π for $spec_{IMMT}$ with a finite t in this case.

Denote by Υ the only fixed point of $\mathcal{M}$. Let v be a correct vertex such that $v \in S_B^*$. By definition of S_B^*, we have: $max_met(g, v, r) \prec max_met(g, v, b)$ for at least one Byzantine vertex b. As $\mathcal{M}$ is strictly decreasing and has only one fixed point, we can deduce that $\Upsilon \preceq max_met(g, v, r)$ and then $max_met(g, v, b) \neq \Upsilon$.

Assume by contradiction that $dist(g, v, b) > c \geq |M(g)| - 2$. As $\mathcal{M}$ is strictly decreasing, has only one fixed point Υ, and $\mathcal{M}$ has $|M(g)|$ distinct metric values over g, we can conclude that $max_met(g, v, b) = \Upsilon$. This contradiction allows us to conclude that there exists a vertex b such that $dist(g, v, b) \leq c$ for any correct vertex which belongs to S_B^*.

In other words, $S_B^* = \left\{v \in V \middle| \underset{b \in B}{min}\{dist(g, v, b)\} \leq c\right\}$ and π is in fact a $(t, c, n-1)$-strongly stabilizing protocol with a finite t, that proves the result.

2) Proof of the "only if" part: This result is a direct consequence of Lemma 15.15 when we observe that $|M(g)| \leq |M|$ by definition. ■

15.4 Summary

Table 15.1 summarizes all results presented in this chapter about maximum metric spanning tree construction in systems subject to any transient and intermittent Byzantine fault pattern. Note that these results generalize all results proved in

Chapter 14 about spanning tree construction and BFS spanning tree construction in the same context.

Chapter 16

Conclusion of Part IV

If I knew I should die tomorrow, I would plant a tree today.

Stephen Girard

Contents

16.1 Summary of Contributions 227
16.2 Concluding Remarks 229

16.1 Summary of Contributions

The fourth part of this thesis focused on maximum metric spanning tree construction in distributed systems subject to any transient and intermittent Byzantine fault pattern. Spanning tree construction is a fundamental task in distributed systems since it permits to construct a virtual communication structure that allows every vertices to communicate using a minimal number of edges of the original communication graph. According to desired characteristics of the spanning tree (minimum weight, shortest path to the root, minimal degree,...), there exists numerous self-stabilizing distributed protocols.

To our knowledge, the work presented here is the first to consider spanning tree construction in presence of both transient and Byzantine faults. As this problem is global (whatever the considered spanning tree properties are), there exists no strictly-stabilizing solutions for any (finite) containment radius by the generic impossibility result of Nesterenko and Arora [NA02]. Therefore, our first contribution was to propose three new schemes of Byzantine containment in self-stabilization in order to by-pass this impossibility result.

First, we proposed strong stabilization, where the constraint about the containment radius is relaxed, *i.e.* there may exist vertices outside the containment radius that invalidate the specification predicate, due to Byzantine actions. However, the impact of Byzantine triggered action is limited in times: the set of Byzantine vertices may only impact vertices outside the containment radius a bounded number of times, even if Byzantine vertices execute an infinite number of actions. This new scheme of Byzantine containment in self-stabilization generalizes strict stabilization as a strictly stabilizing distributed protocol is a strongly stabilizing one with a maximal number of disruptions equal to 0.

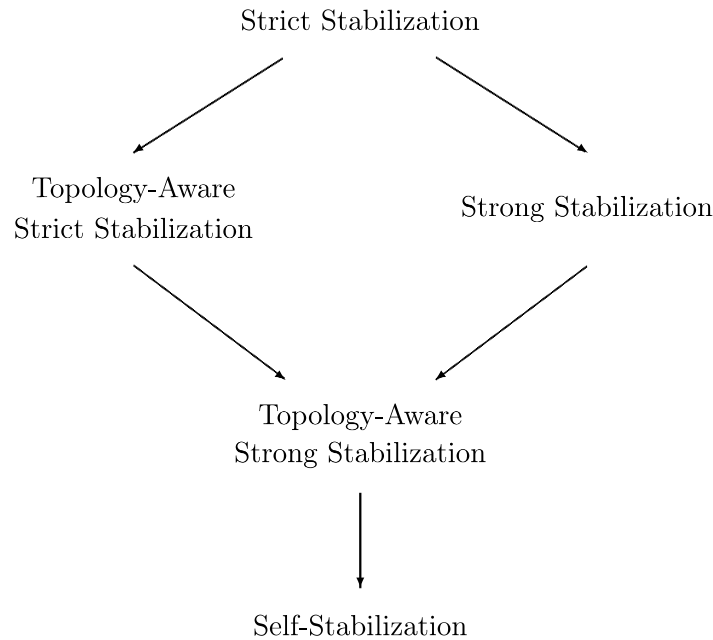

Figure 16.1: Summary of respective constraints on Byzantine containment schemes in self-stabilization. An arrow from a scheme to another means that the first is more constrained than the second.

Although this new scheme is sufficient to by-pass some impossibility results (see *e.g.* Theorem 14.1), it is still too strong for some problems as there remains impossibility results in the context of strong stabilization (see *e.g.* Theorem 14.2). We proposed a new notion for Byzantine containment in self-stabilization: the topology-aware stabilization. Here, the requirement about the containment radius is relaxed to a containment area, *i.e.* the set of vertices which may be disturbed by Byzantine ones is not reduced to the union of c-neighborhood of Byzantine vertices but is defined as a function of the communication graph and Byzantine vertices locations. Note that this relaxation may be applied either to strict or to strong stabilization. Figure 16.1 summarizes comparisons between the three schemes of Byzantine containment in self-stabilization we introduced and strict-stabilization.

To demonstrate the effectiveness of our notions of strong stabilization and topology-aware stabilization, we focused on a large class of spanning tree constructions: the maximum metric spanning tree construction with respect to any maximizable metric. Intuitively, a metric is a scheme to compute a distance along any path of the communication graph. A metric is maximizable if there always exists a spanning tree that maximizes the metric of each vertex of any communication graph with respect to a distinguished vertex called the root. For example, the shortest path or the flow metric are maximizable. In contrast, there exists no maximizable metric to model the minimum weight or the minimum degree spanning tree construction.

In this context, a summary of our contributions follows. First, Chapter 14 proves that all maximizable metrics are not equivalent with respect to Byzantine

	Containment	Result	Proved by...
(c, f)-strict stabilization	$c \in \mathbb{N}, f = 1$	Impossible	[NA02]
(t, c, f)-strong stabilization	$c \in \mathbb{N}, t \in \mathbb{N}$, $f = n - 1$	Possible $\Leftrightarrow \mathcal{C}(\mathcal{M}, c)$	Theorem 15.5
(C_B, f)-TA strict stabilization	$C_B \subsetneq S_B, f = 1$	Impossible	Theorem 15.3
	$C_B = S_B, f = n - 1$	Possible	Theorem 15.1
(t, C_B, f)-TA strong stabilization	$C_B \subsetneq S_B^*, f = 1$	Impossible	Theorem 15.4
	$C_B = S_B^*, f = n - 1$	Possible	Theorem 15.2

Table 16.1: Summary of results of Chapter 15 related to $spec_{IMMT}$ with $\mathcal{C}(\mathcal{M}, c)$ a predicate that is true if and only if $\mathcal{M} = (M, W, mr, met, \prec)$ is a strongly maximizable metric and $c \geq max\{0, |M(g)| - 2\}$.

containment in self-stabilization. Indeed, we prove that there exists a strongly stabilizing solution to the spanning tree construction (without constraints, modeled by the maximizable metric $\mathcal{NC}$) whereas it is impossible to solve the BFS spanning tree construction in a strongly stabilizing way ($\mathcal{BFS}$ is indeed a maximizable metric). We also provide a distributed protocol that solves this last problem in a topology-aware stabilizing way (by providing optimal containment areas with respect to both topology-aware strict and strong stabilization).

Finally, Chapter 15 deals with the problem of maximum metric spanning tree construction for any maximizable metric. Our results are summarized in Table 16.1. We provide a distributed protocol that achieves optimal containment areas with respect to both topology-aware strict and strong stabilization (these areas obviously depend on the considered metric) and a full characterization of maximizable metrics that allow strong stabilization.

16.2 Concluding Remarks

The results presented in this part show that our new notions of strong stabilization and topology-aware stabilization are convenient to by-pass some impossibility results related to strict stabilization but raise some open questions.

A part of these open questions is directly related to our results about maximum metric spanning tree construction. We list them here:

1. In Chapter 15, we choose to work with a slightly modified specification of the problem in order to keep the consistency of results. An interesting question is to provide similar results for the original specification, namely $spec_{MMT}$. We may also try to modify this specification in order to remove *dist* from the set of O-variables. Indeed, this variable is not really necessary to the specification since it is in fact introduced to break cycles.
2. Daemon requirements for the correctness of our distributed protocols is not discussed here. It would be interesting to prove their necessity or to provide distributed protocols with weaker daemon if possible. In this case, is it possible to keep the optimality of containment areas?

3. Another way to complete results of this part is to study the relationship between the containment areas and the maximal number of disruptions. Intuitively, if constraints on containment area are weakened, the maximal number of disruptions may decrease.
4. While maximizable metrics are a large class of metrics, there exists numerous other metrics to construct spanning tree. We think that the study of Byzantine containment properties of these metrics could be interesting to study.

The other part of these open questions is more general and is related to our new scheme of Byzantine containment in self-stabilization. Note that all distributed protocols provided in this part that achieves strong-stabilization (or topology-aware strong stabilization) needs (at least) a strongly fair daemon. We conjecture that the strong fairness property is necessary to perform (topology-aware) strong stabilization. The proof of this conjecture is an interesting open question.

Finally, we presented definitions for strong stabilization and topology-aware stabilization for static problems, *i.e.* problems that require the system to find static solutions, as the maximum metric spanning tree construction. An interesting path for future research may be to provide strong or topology-aware stabilizing distributed protocols for other static problems or to extend our definitions to dynamic problems such as token circulation or leader election.

Chapter 17
Conclusion

The whole of science is nothing more than a refinement of everyday thinking.

Albert Einstein

Contents

17.1 Overview of Thesis Contributions **231**
17.1.1 Part One: Context 231
17.1.2 Part Two: Atomic Register 232
17.1.3 Part Three: Unison 233
17.1.4 Part Four: Spanning Tree 233
17.1.5 Summary 234
17.2 Perspectives **234**

This concluding chapter surveys thesis contributions (see Section 17.1) and discusses questions raised by our works (see Section 17.2).

17.1 Overview of Thesis Contributions

The main problem we study in this thesis is the joint tolerance to transient faults and to permanent or intermittent faults in distributed systems. The main difficulty of such an environment is the unabililty for correct vertices to distinguish malicious (or crashed) vertices from honest but badly initialized ones. This situation allows permanent or intermittent faulty vertices to disturb (that is, to prevent satisfying a particular specification) some correct vertices. The challenge is then to design distributed protocols that achieve the best possible fault tolerance or containment in such a context.

In the sequel, we describe more precisely the contributions of each part of this thesis.

17.1.1 Part One: Context

After describing our model for distributed systems, the first part of this thesis proposed a taxonomy of daemons. A daemon is a classical abstraction in self-stabilization that gathers assumptions related to the scheduling of the distributed system. We show that daemons can be compared using a partial order. A daemon is

more powerful than another one if it allows strictly more executions. When designing a distributed protocol, it is desirable to deal with the most powerful daemon that still enable problem solvability, as it allow the protocol to run properly in a larger set of real environments. Our taxonomy allows distributed protocol designers to easily compare their assumptions about scheduling.

The last chapter of this part is devoted to a survey of variants of self-stabilization. First, a self-stabilizing distributed protocol ensures that, starting from any arbitrary initial configuration (that is the consequence of transients faults), a correct behavior is recovered in a finite time. It is well known that self-stabilization is a non masking approach of fault tolerance since an external observer may see the effects of transient faults during a finite time. Our survey mainly focuses on variants of self-stabilization that are moreover able to tolerate a limited number of permanent or intermittent faults after the end of transient faults. In particular, we present the two fault tolerance schemes used in this thesis: fault-tolerant pseudo-stabilization and strict stabilization. The first one ensures that, starting from any arbitrary configuration, any (infinite) execution has an infinite suffix that satisfies the specification in spite of a given number of permanent crashes. The second one ensures that, starting from any arbitrary configuration, any execution reaches in a finite time a configuration from which the effects of a given number of intermittent Byzantine faults are contained within a given radius around them. Note that fault-tolerant pseudo-stabilization has a masking approach with respect to permanent faults while strict stabilization has a non masking approach with respect to intermittent Byzantine faults.

17.1.2 Part Two: Atomic Register

Self-stabilizing or fault-tolerant distributed protocols are often designed in a high atomicity computational model due as they are simpler to write proof with compared to a lower atomicity one (such as the message passing model). This motivates a field of research that consists in designing computational model transformers. These distributed protocols simulate a high atomicity computational model over a low atomicity one in order to allow a simpler design of distributed protocols and their correctness proof.

The second part of this thesis focuses on the simulation of a high atomicity computational model that is extensively used in fault tolerant distributed computing: the atomic register model. More precisely, we provide a fault-tolerant pseudo-stabilizing single-writer multi-reader atomic register simulation. This simulation is the pseudo-stabilizing version of the classical bounded ABD simulation.

In order to adapt the ABD simulation to tolerate transient faults, we designed two tools (that are sufficiently general and independent to be re-used in another context). The first one is a self-stabilizing communication primitive between two neighboring vertices over non reliable and non FIFO communication links of bounded capacity. We define for the first time a set of metrics that allows to quantify the impact of transient faults on the communication primitive and we prove that our

solution is optimal with respect to each of these metrics. The second tool is a sequential bounded labeling scheme that is able to cope with any arbitrary initial configuration (conversely to existing ones).

17.1.3 Part Three: Unison

The third part of this thesis focuses on the synchronization problem. We assume that each vertex has a digital clock and that we need some synchronization between these clocks in presence of both transient and intermittent Byzantine faults. As it is impossible to strongly synchronize the clocks in an asynchronous distributed systems in such a context, we consider the unison problem (that is, self-stabilizing weak clock synchronization). Intuitively, clock drift between neighboring vertices must be eventually bounded by one and each vertex updates infinitely often its clock by incrementing it. Due to the possibility of deadlocks induced by Byzantine vertices, we must weaken this last part of the specification by ensuring only that every clock of a correct vertex is incremented infinitely often. We then define two particular classes of unison. A minimal unison admits only one variable: the clock. A priority unison ensures a property similar to obstruction-freedom. Note that any self-stabilizing unison we are aware of falls in at least one of these two classes.

First, we prove a broad class of impossibility results related to this problem in our environment. We prove the impossibility of the problem when there are two or more faulty vertices or when the daemon is unfair. We also prove the impossibility of minimal or priority unison when the daemon is weakly fair and when the daemon is strongly fair and the communication graph is not reduced to a chain or a ring.

On the positive side, we provide a strictly stabilizing minimal and priority unison protocol for the remaining cases (that is, this protocol tolerates at most one Byzantine fault, is designed for communication graphs reduced to chains or rings, and requires a strongly fair daemon). This protocol is thus proved optimal with respect to impossibility results. Moreover, we prove that it is also optimal with respect to convergence time.

17.1.4 Part Four: Spanning Tree

Spanning tree construction is a fundamental building block in many distributed protocols since a spanning tree ensures connectivity using a minimal number of communication links. In the fourth part of this thesis, we focus on a generic distributed protocol that constructs a maximum metric spanning tree with respect to any maximizable metric. Intuitively, a metric is a scheme to compute a distance along any path of the communication graph. A metric is maximizable if there always exists a spanning tree that maximizes the metric of each vertex of any communication graph with respect to a distinguished vertex.

As this problem is global, it is impossible to provide a strictly stabilizing solution for any containment radius. Therefore, we introduce three new Byzantine containment schemes in self-stabilization to bypass this impossibility result. First,

we propose strong stabilization in which the constraint to the containment radius is relaxed, *i.e.* there may exist vertices outside the containment radius that invalidate the specification, due to Byzantine actions. However, the impact of Byzantine triggered action is limited in times: the set of Byzantine vertices may only impact vertices outside the containment radius a bounded number of times, even if Byzantine vertices execute an infinite number of actions. Then, we propose another notion for Byzantine containment in self-stabilization: the topology-aware stabilization. Here, the requirement to the containment radius is relaxed to a containment area, *i.e.* the set of vertices which may be disturbed by Byzantine ones is not reduced to the union of c-neighborhood of Byzantine vertices but is defined as a function of the communication graph and Byzantine vertices location. Note that this relaxation may be applied either to strict or to strong stabilization.

After a detailed study of two particular maximizable metrics, we provide a maximum metric spanning tree construction distributed protocol for any maximizable metric. We prove that this distributed protocol exhibits optimal Byzantine containment with respect to both topology-aware strict and strong stabilization. Finally, we focus on strong stabilization. We characterize the set of maximizable metrics that allow the existence of a strongly-stabilizing distributed protocol.

17.1.5 Summary

As a summary of contributions of this thesis, we propose to complete Figure 4.1 (that sums up the respective constraints on permanent or intermittent fault tolerant schemes in self-stabilization) by adding fault-tolerance schemes introduced in this thesis.

In this way, Figure 17.1 compares all fault-tolerance schemes that appear in this thesis. Recall that we say that a fault tolerance scheme is more constrained that another if any distributed protocol that satisfies the first satisfies the second.

17.2 Perspectives

Immediate research perspectives were already discussed in the concluding chapter of each contributing part. We now present general mid or long term research goals raised by our work.

New fault-tolerance schemes This thesis focused on stabilizing distributed protocols that are moreover able to contain the effects of a given class of permanent or intermittent faults (that are supposed to occur after the end of transient faults). An interesting path for future research is to study the effectiveness of the combination of variants of stabilization with such containments. For instance, is it possible to provide a snap-stabilizing distributed protocol that is also able to contain the effects of intermittent Byzantine faults?

In general, the introduction of new fault-tolerance schemes would permit to enrich the available panel of fault-tolerant solutions since the distributed system

designers have to manage an intricate trade-off between:

1. the strength (with respect to fault tolerance or fault containment) of the chosen fault-tolerance scheme,
2. the scope of problems that remain solvable using such a scheme,
3. the cost of this scheme for a particular problem,
4. the criticality of the application, and
5. the frequency of fault occurrences in the system.

For instance, it is useless to guarantee very strong fault-tolerance properties at a high cost (with respect to used resources such as memory or bandwidth) for a routing distributed protocol in a network that experiences faults once a year. On the other hand, if human life depends on the correctness of the application (*e.g.* air traffic control), strong fault-tolerance is desirable whatever the cost is.

Bounding the power of Byzantine vertices Another way to extend results presented in this thesis is to restrict the model we used for Byzantine faults. For example, we considered only Byzantine vertex with an unbounded power (that is, they may execute an infinite number of malicious actions). Another possible model of Byzantine faults may consider bounded power (that is, each Byzantine vertex can perform only a finite number of malicious actions in any execution, as these actions have a high cost for the Byzantine node). Some interesting questions can be considered in this model: Is there a trade-off between the number of Byzantine actions and the containment radius/area? Is there a trade-off between the total number of perturbations Byzantine vertices can cause and the number of Byzantine vertices, that is, is a single Byzantine vertex more effective to harm the distributed system than a team of Byzantine vertices, considering the same total number of Byzantine actions?

Once again, the goal is to provide a large panel of distributed protocols with various fault-tolerance properties in order that the user can choose the most convenient for his application and his particular constraints. Indeed, it is useless to provide "perfect" fault tolerance to unbounded Byzantine faults at high cost if the considered distributed system is only hit by message omission (for which it may be possible to design an *ad hoc* solution at a lower cost). Defining a rigorous taxonomy of faults that allows comparison of the relative power of fault tolerance schemes is an interesting challenge.

Probabilistic approach In this thesis, we chose to only study deterministic issues related to permanent or intermittent fault containment in self-stabilization. Another way to bypass numerous impossibility results presented in this thesis is to guarantee the stabilization of the distributed system only with a high probability.

As in self-stabilization, we think that it is possible to define several notions of probabilistic permanent or intermittent fault tolerance in self-stabilization. For instance, we can define a fault-tolerance scheme that ensures properties that are

similar to the ones of strict stabilization but with a bounded expected convergence time only (instead of a bounded convergence time) or a probabilistic closure. Strong stabilization may be extended by guaranteeing a bounded expected number of disruptions. The main question is then to evaluate the benefits of randomization on containment radius or areas with respect to deterministic solutions. In other words, the issue is still to allow the final user of the distributed system to find the best trade-off between fault-tolerance properties, implementation difficulty, cost of distributed protocols, and impossibility results for its application.

Alternate distributed system models An appealing way to extend results presented in this thesis is to consider another models of distributed systems. We can foresee at least two paths of research:

1. We may consider models that includes various new assumptions. Numerous models of such new distributed systems have been described in the past few years. For instance, one can consider dynamic distributed systems in which communications links may appear or disappear over time, sensor systems in which vertices communicate by radio medium and are limited by their battery capacities, or robots systems in which vertices are enhanced with mobility capacities. The study of simultaneous tolerance to several kind of faults in such systems is a challenging task.
2. We may also consider more realistic models for distributed systems retaining similar constraints with respect to fault or attack tolerance to the ones presented in this thesis. An interesting open question is then the design of automatic transformers for such models that ensure fault-tolerance properties preservation of the original distributed protocol.

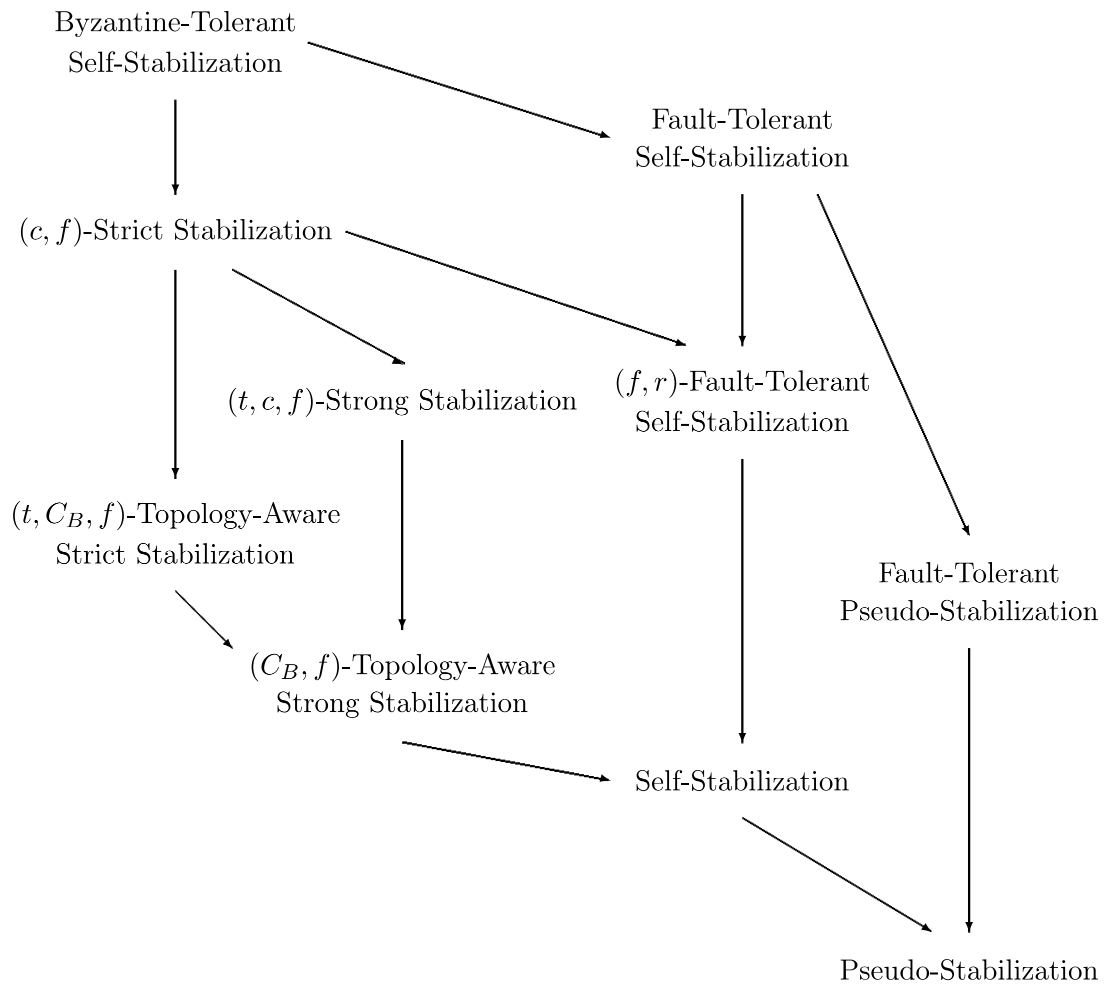

Figure 17.1: Summary of respective constraints on fault-tolerance schemes used in this thesis. An arrow from a scheme to another means that the first is more constrained than the second. Note that we remove all arrows deductible from transitivity.

Annexe A
Version française

Le vrai danger, ce n'est pas quand les ordinateurs penseront comme les hommes, c'est quand les hommes penseront comme les ordinateurs.

Sydney J. Harris

Sommaire

A.1 Contexte de la thèse **240**
A.1.1 Généralités 240
A.1.2 Modèles et tolérance aux fautes 243
A.2 Registre atomique **245**
A.2.1 Contexte 245
A.2.2 Contributions 246
A.2.3 Perspectives 246
A.3 Unisson **247**
A.3.1 Contexte 247
A.3.2 Contributions 248
A.3.3 Perspectives 248
A.4 Arbre couvrant **249**
A.4.1 Contexte 249
A.4.2 Contributions 250
A.4.3 Perspectives 251
A.5 Conclusion **251**

Cette annexe constitue un résumé des principales contributions de cette thèse. Ce résumé reste volontairement à un niveau purement intuitif car le lecteur intéressé par la présentation complète et rigoureuse d'une contribution sera renvoyé au chapitre correspondant de la thèse.

Dans un premier temps, nous présentons la problématique de cette thèse à l'aide d'une illustration ne relevant pas de l'Informatique.

Longtemps après sa victoire contre le Seigneur des Anneaux Sauron [Tol37, Tol54a, Tol54b, Tol55], Frodon Sacquet se perdit dans un immense labyrinthe perdu au fin fond de la Comté. Lorsque Sam Gamegie réalisa sa disparition, il demanda à tous les Hobbits de l'aider à sauver Frodon. Ils rentrèrent alors tous dans le labyrinthe et se dispersèrent autant que possible. Les Hobbits étaient suffisament

nombreux pour couvrir le labyrinthe en entier (il y avait au moins un Hobbit par intersection). Chaque Hobbit n'était capable de communiquer qu'avec ceux situés aux intersections avoisinantes en raison du bruit qu'ils provoquaient dans le labyrinthe. Leur but était d'indiquer à Frodon le chemin de la sortie en lui indiquant la route à chaque intersection. Cependant, comme il est bien connu que les Hobbits aiment la bière et le cidre, ils étaient tous ivres quand ils entrèrent dans le labyrinthe et ils s'endormirent tous en arrivant à leur position. Lorsqu'ils se réveillèrent quelques heures plus tard, chacun d'eux était désorienté et indiquait une direction arbitraire à Frodon. En revanche, même s'ils étaient désorientés, les Hobbits voulaient toujours aider Frodon à sortir. Dans ce but, ils collaborèrent pour recouvrer de leur désorientation. Le Hobbit qui était situé à la sortie du labyrinthe en informa ses voisins. Ceux-ci choisirent d'indiquer ce chemin à Frodon et propagèrent l'information. Comme tous les Hobbits firent de même, ils furent capable d'indiquer un chemin complet vers la sortie à Frodon quelle que soit sa position initiale dans le labyrinthe.

Imaginons à présent un autre scénario dans lequel certains des Hobbits qui entrèrent dans le labyrinthe sont en réalité des traîtres qui tentaient de venger Sauron en perdant Frodon à tout jamais dans ce labyrinthe. Lorsque les Hobbits honnêtes se réveillèrent en étant désorientés, ces traîtres purent leur mentir afin de les empêcher d'atteindre leur objectif. Ces Hobbits menteurs n'avaient alors plus le même objectif que les autres Hobbits : ils voulaient empêcher Frodon d'atteindre la sortie alors ques les Hobbits honnêtes souhaitaient qu'il sorte. Les Hobbits menteurs prétendirent être situés à la sortie du labyrinthe même si çà n'était pas le cas. Les Hobbits honnêtes n'eurent alors pas d'autre choix que de propager cette information comme précédement (ils n'avaient aucun moyen de déterminer si cette information est vraie ou pas). Les Hobbits menteurs peuvent ainsi attirer Frodon vers eux si il était initiallement plus proche de l'un d'eux que de la vraie sortie du labyrinthe.

Sur cet exemple intuitif, le cœur de cette thèse consiste à étudier l'impact de la combinaison de la désorientation initiale des Hobbits honnêtes et des mensonges des traîtres sur les capacités des Hobbits honnêtes à atteindre leur objectif (dans notre exemple, indiquer le chemin de la sortie à Frodon).

Dans la suite de cette annexe, nous présentons le contexte de cette thèse en expliquant l'analogie avec notre illustration (voir section A.1) puis nous détaillons les contributions de cette thèse dans les sections A.2, A.3 et A.4. Nous concluons dans la section A.5.

A.1 Contexte de la thèse

A.1.1 Généralités

Systèmes répartis et tolérance aux fautes L'algorithmique répartie est une branche de l'Informatique qui étudie les systèmes répartis. Intuitivement, un système réparti est un système constitué d'unités de calcul autonomes (appelées processeurs ou processus) dotées de capacités de communication. Chaque processeur peut communiquer avec un sous ensemble des autres processeurs. De ce fait, il est naturel

de représenter les possibilités de communication d'un tel système par un graphe. Les principales caractéristiques de ce type de système sont la localité de l'information (chaque processeur possède seulement une vue locale sur le système et doit communiquer avec les autres processeurs pour obtenir n'importe quelle information globale) et la localité du temps (chaque processeur exécute ses instructions à son propre rythme). Ce modèle est suffisament général pour englober les caractéristiques de n'importe quel type de réseau (réseau local, réseau de capteurs, système pair-à-pair, ...). Dans notre illustration, les Hobbits disséminés dans le labyrinthe peuvent être vus comme un système réparti. En effet, nous pouvons représenter le labyrinthe par un graphe (chaque sommet représente une intersection) et les Hobbits peuvent être vus comme des unités de calcul capable de communiquer avec celles situées aux intersections limitrophes.

Le but de l'algorithmique répartie est la conception de protocoles résolvant certains problèmes dans les systèmes répartis. De tels protocoles sont appelés protocoles répartis. Générallement, la difficulté provient du fait que les problèmes considérés portent sur des propriétés globales du système. En conséquence, les processeurs ne peuvent pas les résoudre de manière locale et doivent communiquer et coopérer pour atteindre un objectif commun. Dans notre illustration, les Hobbits du labyrinthe ont un problème global à résoudre : trouver un chemin menant Frodon à la sortie. Comme aucun Hobbit n'a de solution locallement, ils doivent coopérer pour trouver un tel chemin. En systèmes répartis, ce problème est connu comme le problème de construction d'arbre couvrant. Un processeur donné est distingué comme la racine du système (la sortie du labyrinthe) et chaque processeur doit choisir un de ses voisins comme parent (le premier processeur du chemin entre lui et la racine). Dans notre illustration, il s'agit pour chaque Hobbit d'indiquer le chemin de la sortie à Frodon.

Lorsque la taille d'un système réparti augmente ou lorsqu'il est déployé dans un environnement dangereux, nous ne pouvons pas négliger la probabilité que certains éléments du système se comportent incorrectement (par exemple, les communications peuvent interférer, certains processeurs peuvent arrêter d'exécuter leurs instructions, être sujets à des attaques ou des virus, ...). Tout comportement anormal de tout élément d'un système réparti est modélisé par le concept de faute. Comme ce concept est très général, il est classiquement admis de classifier les fautes selon plusieurs critères. Par exemple, la durée de la faute peut être prise en compte. Nous distinguons une faute transitoire (une faute de durée finie) d'une faute permanente (faute de durée infinie) ou d'une faute intermittente (les processeurs affectés exhibent successivement des comportement corrects et incorrects). Nous pouvons également distinguer les fautes par leur nature. Par exemple, nous pouvons considérer des fautes crashs (les processeurs affectés cessent d'exécuter leurs instructions), des fautes Byzantines (les processeurs affectés exhibent un comportement arbitraire) ou des corruptions de mémoire... Dans notre illustration, l'ivresse des Hobbits peut être interprétée comme une faute transitoire (les Hobbits se réveillent en un temps fini) induisant une corruption de mémoire (les Hobbits sont désorientés à leur réveil) tandis que la trahison de certains Hobbits peut être vue comme une faute Byzan-

tine permanente (ces Hobbits ne coopérent plus avec les autres à la réalisation de l'objectif commun).

La tolérance aux fautes répartie est une sous-branche de l'algorithmique répartie qui se concentre sur les protocoles répartis tolérant des fautes. En d'autres mots, un protocole réparti tolérant aux fautes assure que certaines propriétés sont garanties même si certaines fautes touchent le système. Il existe plusieurs notions de tolérance aux fautes en fonction de la classe de fautes tolérées. Dans la suite, nous présentons les notions de tolérance aux fautes étudiées dans cette thèse.

Auto-stabilisation Dans notre première illustration dans laquelle tous les Hobbits sont honnêtes, ils parviennent ultimement à indiquer à Frodon un (plus court) chemin vers la sortie même s'ils étaient initialement désorientés. Cette capacité à retrouver un comportement correct en un temps fini à partir d'un état initial arbitraire est appellé auto-stabilisation en algorithmique répartie. L'auto-stabilisation [Dij74] permet de tolérer des fautes transitoires (quelle que soit la nature de la faute). En effet, l'état du système peut être quelconque à la fin d'une faute transitoire (en raison des actions anormales durant la faute, de corruption de mémoire, ...). Un protocole réparti auto-stabilisant assure alors que le système retrouvera un comportment correct en un temps fini sans aide extérieure ou manuelle. Il est important de noter qu'un protocole réparti auto-stabilisant ne peut tolérer que des fautes transitoires. En effet, un tel protocole est basé sur l'hypothèse que les instructions de chaque processeur ne sont pas corrompues par la faute transitoire (seule la mémoire volatile l'est) et que chaque processeur exécute correctement son protocole après la fin de la faute transitoire. Dans le cas contraire, le protocole risque de ne jamais retrouver un comportement correct.

Confinement de fautes crashs/Byzantines en auto-stabilisation Pour assurer une meilleure tolérance aux fautes, il est possible de considérer des protocoles répartis auto-stabilisants qui sont de plus capables de gérer un nombre limité de fautes permanentes ou intermittentes après la fin des fautes transitoires. En raison de ce modèle de fautes très ambitieux à contrôler, nous ne pouvons pas garantir que le système entier retrouve un comportement correct en un temps fini comme en auto-stabilisation (au moins les processeurs fautifs peuvent exhiber un comportement anormal infiniment longtemps). Dans notre seconde illustration dans laquelle il existe des traîtres qui veulent perdre Frodon dans le labyrinthe, le protocole réparti utilisé par les Hobbits honnêtes garantit seulement que Frodon trouvera la sortie s'il est initiallement (strictement) plus proche de la sortie que d'un Hobbit menteur. Cette propriété de tolérance aux fautes peut sembler faible au premier abord mais nous prouvons dans cette thèse que nous ne pouvons pas en garantrir une plus forte si les Hobbits honnêtes sont initialement désorientés.

Le principal problème abordé dans cette thèse est la tolérance conjointe aux fautes transitoires et aux fautes permanentes ou intermittentes dans les systèmes répartis. La principale difficulté provient du fait suivant : dans de tels systèmes, il

est impossible de distinguer un processeur fautif (de manière permanentes ou intermittente) qui ne collabore plus à la réalisation de l'objectif commun du système d'un processeur mal initialisé mais honnête (qui coppére pour atteindre l'objectif global du système en dépit des fautes transitoires). Dans notre illustration, les traîtres peuvent attirer Frodon dans la mauvaise direction car les Hobbits honnêtes n'ont aucun moyen de distinguer le Hobbit honnête situé à la sortie du labyrinthe d'un Hobbit menteur et propagent donc les deux informations de la même manière.

A.1.2 Modèles et tolérance aux fautes

La première partie de cette thèse est consacrée à la présentation détaillée des modèles utilisés ainsi que la définition de certains concept classiques de tolérance aux fautes. Ces présentations sont accompagnées d'états de l'art détaillés.

Modèles Le chapitre 2 présente les modèles de calcul utilisés dans cette thèse ainsi que le modèle de faute adopté. Le système réparti est classiquement représenté par un graphe de communication dans lequel les sommets représentent les processeurs tandis que les arêtes représentent les liens de communications entre processeurs. Nous présentons ensuite les deux modèles de calcul utilisés dans cette thèse. Le premier est le modèle à états dans lequel chaque processeur peut, en une étape atomique (*i.e.* indivisible) , lire l'état de tous ses voisins et modifier son propre état en fonction de cette lecture. Le second est le modèle par passage de message dans lequel chaque processeur peut, en une étape atomique, envoyer ou recevoir un message à un des ses voisins ou exécuter une instruction interne. Le premier modèle est largement utilisé en auto-stabilisation bien qu'il soit moins réaliste que le second.

Le chapitre 3 est consacré à la définition formelle d'un concept central de l'auto-stabilisation : le démon [Dij74]. Il s'agit d'une abstraction permettant de modéliser l'asynchronisme du système réparti. Il peut être vu intuitivement comme un ordonnanceur qui sélectionne un sous-ensemble de processeurs à chaque étape. Ces processeurs séléctionnés sont alors autorisés à exécuter leur protocole. Nous identifions quatre caractéristiques permettant de classifier les démons :

1. La distribution du démon caractérise la distance minimale entre deux processeurs simultanément choisis par le démon.
2. L'équité du démon caractérise les choix du démon dans le temps.
3. La borne d'un démon est le nombre maximal d'activations d'un processeur autorisées par le démon entre deux activations consécutives d'un autre processeur.
4. L'activabilité d'un démon est le nombre maximal d'étapes dans lesquelles un processeur n'est pas sélectionné par le démon entre deux de ses actions consécutives.

Nous établissons un ordre partiel permettant de comparer les démons entre eux. Cet outil est utile pour comparer entre eux des protocoles résolvant le même problème

mais sous des démons différents. Enfin, nous proposons un état de l'art des transformateurs de démons existants. Ces protocoles permettent à un protocole réparti de s'exécuter sous un démon plus fort tout en préservant l'auto-stabilisation.

Tolérance aux fautes Le chapitre 4 dresse un état de l'art des concepts de tolérance aux fautes dans les systèmes répartis. Il existe deux critères principaux pour les distinguer les uns des autres. Le premier est la classe de fautes tolérées tandis que le second est la propriété de masquage [Tix09]. Un concept de tolérance aux fautes est dit masquant lorsqu'il assure que les fautes que subit le système réparti ne sont pas visibles pour un observateur extérieur.

Par exemple, un protocole robuste aux crashs (qui guarantit que le système se comporte normalement même en présence d'un nombre limité de fautes crashs) peut être qualifié de masquant étant donné qu'un observateur extérieur ne peut pas savoir si une exécution donnée a subi des crashs ou non. En revanche, l'auto-stabilisation n'est pas masquante car un observateur extérieur peut constater un comportement anormal du système durant la phase de stabilisation.

Nous présentons maintenant les variantes de l'auto-stabilisation qui sont utilisées dans cette thèse :

Pseudo-stabilisation : La pseudo-stabilisation [BGM93] est une variante affaiblie de l'auto-stabilisation étant donné que nous imposons seulement l'existence d'un suffixe de l'exécution satisfaisant la spécification (et non plus l'existence d'un préfixe borné ne respectant pas la spécification).

Ce concept plus faible peut être utile par exemple lorsque l'auto-stabilisation est impossible.

Auto-stabilisation tolérante aux fautes : Il s'agit d'une variante de l'auto stabilisation capable de tolérer de plus un nombre limité de fautes crashs [AH93, GP93]. Un protocole réparti est dit auto-stabilisant et tolérant aux fautes s'il guarantit, partant d'un état initial arbitraire, que le système réparti retrouve un comportement correct en un temps fini malgré un nombre borné de fautes crashs.

Il est bien sûr possible de définir une variante de la pseudo-stabilisation procurant des propriétés similaires [DGDF10].

Auto-stabilisation tolérante aux Byzantins : Il s'agit d'un concept de tolérance aux fautes analogue au précédent en considérant des fautes Byzantines au lieu des fautes crashs [DW95].

On peut noter que ces approches sont masquantes pour les fautes permanentes ou intermittentes mais non masquantes pour les fautes transitoires.

Stabilisation stricte : Dans un système réparti sujet à des fautes transitoires arbitraires et à un nombre borné de fautes Byzantines intermittentes, la stabilisation stricte [NA02] garantit que le système atteint en un temps fini une configuration à partir de laquelle seuls les processeurs situés en dessous d'une distance donnée d'un processeur Byzantin ne respectent pas leur spécification.

En d'autres termes, l'effet des fautes Byzantines est confiné en un temps fini autour des processeurs Byzantins.

A.2 Registre atomique

A.2.1 Contexte

Dans la seconde partie de cette thèse, nous nous concentrons sur un problème fondamental des systèmes répartis : la simulation d'un modèle de calcul sur un autre. Le principal intérêt de ce problème vient de l'observation suivante : plus l'atomicité d'un modèle de calcul est forte, plus le dévellopement d'une application répartie dans ce modèle est simple mais, en revanche, moins le protocole obtenu est réaliste. Afin de pouvoir conserver la simplicité d'un modèle de calcul à forte atomicité dans un modèle à atomicité fine, une solution intéressante consiste à dévelloper un transformateur du premier vers le second. De cette manière, nous pouvons dévelloper des protocoles répartis dans le modèle à forte atomicité (en profitant de sa simplicté) et l'exécuter dans le modèle de calcul à atomicité fine (en profitant de son réalisme) en le composant avec le transformateur.

Par exemple, [DIM97a] étudie la simulation du modèle à états sur le modèle à passage de messages dans un contexte auto-stabilisant. Il existe aussi des transformateurs pour des modèles de calcul très spécifiques vers des modèles plus classiques afin de pouvoir ré-utiliser des protocoles répartis existants (écrits pour le modèle classique) dans ces modèles spécifiques. Par exemple, [KA06] propose un transformateur d'un modèle de calcul spécifique aux systèmes répartis sans fils (le modèle de diffusion locale avec collisions) vers le modèle à états.

La principal inconvénient de ces transformateurs de modèles de calcul est qu'ils introduisent généralement un surcoût important (en temps et/ou en mémoire) par rapport à un protocole réparti dévellopé directement dans le modèle de calcul à atomicité fine. En revanche, ils permettent de prouver des équivalences entre les modèles de calcul étant donné que tout problème ayant une solution dans le modèle à atomicité forte en a une dans celui à atomicté fine grâce au transormateur idoine.

Dans cette partie, nous nous concentrons sur la simulation d'un modèle de calcul classiques des systèmes répartis tolérant aux fautes, le modèle à registres partagés (*cf.* [Lyn96]), sur le modèle à passage de messages. Le modèle à registres partagés est caractérisé par le fait que les communications entre les processeurs ont exclusivement lieu à travers des lectures et écritures sur des variables globales partagées (appelées registres). Nous nous intéressons à la classe de registres assurant les plus fortes propriétés, les registres atomiques [Lam86a, Lam86b]. Cette simulation a été très étudiée dans des systèmes répartis sujets à des fautes crashs ou Byzantines.

L'objectif de cette partie est d'étudier la simulation d'une registre atomique sur le modèle à passage de messages dans un système réparti simultanéménent sujet à des fautes transitoires et à des fautes crashs. Le chapitre 5 présente le détail du contexte de cette partie.

A.2.2 Contributions

Les contributions de cette partie se répartissent en deux catégories : l'étude de deux outils indépendants nécessaires à notre simulation et la simulation elle-même.

Nous présentons tout d'abord les deux outils nécessaires à notre simulation. Ces résultats sont détaillés dans le chapitre 6.

1. Nous proposons un protocole de communication permettant de s'affranchir des caractéristiques des canaux de communication. En effet, nous étudions l'effet des fautes transitoires sur des canaux de communication bornés, non fiables et non-FIFO. Le but est de donner un protocole permettant à un processeur d'envoyer des messages à un de ses voisins tout en limitant le nombre de messages perdus, fantômes (*i.e.* non envoyés), dupliqués ou ré-ordonnancés. Nous montrons qu'il est impossible dans de telles circonstances d'empêcher la livraison d'un message fantôme, la duplication d'un message et le ré-ordonnancemnt d'un message. Nous fournissons ensuite un protocole fournissant des performances optimales par rapport à ces quatres critères.
2. De nombreuses simulations de registres reposent sur l'idée d'estampiller les valeurs du registre afin de pouvoir les dater et ainsi retrouver la dernière valeur écrite dans le registre. Il existe principalement deux approches. La première consiste à utiliser des entiers naturels qui garantissent un ordre total mais introduisent une mémoire non bornée. C'est pourquoi la seconde approche priviligie des systèmes d'estampillage bornés. Nous proposons un tel système d'estampillage borné capable de supporter des fautes transitoires.

Le chapitre 7 s'attache à la construction d'une simulation de registre atomique en présence de fautes transitoires et de crashs. Attiya, Bar-Noy et Dolev [ABND95] ont proposé une simulation de registres en mémoire bornée dans des systèmes répartis sujets à une minorité de crashs. Nous montrons dans ce chapitre que ce protocole ne nécessite que peu de modifications (utilisation du protocole de communication et du système d'estampillage du chapitre 6 et détection des estampilles en conflit) pour devenir pseudo-stabilisant et tolérant aux fautes.

Les résultats du chapitre 6 forment un article dans Information Processing Letters [DDPBT11b], une communication dans les actes du 24th International Symposium on Distributed Computing (DISC 2010) [AAD+10] et des 13èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (Algotel 2011) [DDPBT11a]. Concernant le chapitre 7, des versions préliminaires ont été présentées dans les actes du 24th International Symposium on Distributed Computing (DISC 2010) [AAD+10] et du 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011) [AAD+11].

A.2.3 Perspectives

Les travaux présentés dans cette partie ouvrent plusieurs persepectives que nous détaillons dans le chapitre 8 :

1. Notre approche étant modulaire, il est possible d'utiliser le protocole de communication ou le système d'estampillage du chapitre 6 dans d'autres contextes.
2. Il existe de nombreux protocoles répartis tolérant aux crashs écrits dans le modèle à registres partagés. Il serait intéressant de fournir un transformateur permettant de produire leur équivalent auto-stabilisant et tolérant aux fautes (que nous pourrions alors utiliser dans le modèle à passage de messages grâce à notre transformateur).
3. Enfin, nous conjecturons l'impossibilité de produire un simulateur de registre atomique auto-stabilisant et tolérant aux fautes dans notre contexte. La preuve de cette conjecture montrerait la nécessité de notre approche pseudo-stabilisante.

A.3 Unisson

A.3.1 Contexte

La troisième partie de cette thèse s'intéresse à un problème classique des systèmes répartis : la synchronisation. L'intérêt de la synchronisation réside dans l'observation suivante : dans un système réparti sujet à des fautes, plus les propriétés de synchronisation du système sont fortes, plus le dévellopement d'une application répartie est simple [FLP85]. En effet, Fisher, Lynch et Patterson ont démontré l'impossibilité du consensus dans un système réparti asynchrone sujet à des fautes permanentes alors que ce problème est facilement résolvable dans un système réparti synchrone.

C'est pourquoi une solution intéressante pour dévelloper un protocole réparti dans un système asynchrone consiste à d'abord dévelloper un synchronisateur [Mis91] (un protocole réparti qui assure certaines propriétés de synchronisation dans un système asynchrone), de dévelloper un protocole réparti synchrone pour le problème initial et finalement de composer ces deux protocoles afin d'obtenir un protocole réparti pour le problème dans un système asynchrone.

Plus précisèment, nous considérons dans cette partie le problème de l'unisson asynchrone [GH90, CFG92] qui requiert que tous les processeurs maintiennent une synchronisation entre leurs compteurs appelés horloges logiques. Chaque processeur est tenu d'incrémenter infiniment souvent d'une unité son horloge logique tout en maintenant une différence d'horloge maximale d'une unité avec chacun de ses voisins. L'unisson asynchrone est une brique de base fondamentale de plusieurs protocoles répartis.

L'objectif de cette partie est l'étude de l'unisson asynchrone dans un contexte nouveau. Plus précisèment, nous tenons d'apporter une solution à ce problème dans un système réparti simultanément sujet à des fautes transitoires et à des fautes Byzantines intermittentes. Le chapitre 9 détaille le contexte de cette partie. En particulier, nous y identifions deux propriétés des unissons asynchrones auto-stabilisants connus : la minimalité et la priorité.

A.3.2 Contributions

La première contribution de cette partie est la généralisation de la définition de l'auto-stabilisation tolérante aux fautes en utilisant le concept du rayon de confinement de la stabilisation stricte. Cette généralisation est motivée par l'observation suivante : un résultat d'impossibilité pour l'auto-stabilisation tolérante aux fautes avec un rayon de confinement donné implique un résultat similaire pour la stabilisation stricte avec le même rayon de confinement.

En effet, le chapitre 10 est consacré à une série de résultats d'impossibilité pour l'unisson asynchrone dans un contexte auto-stabilisant tolérant aux fautes. Nous montrons en particulier qu'il est impossible de dévelloper un protocole d'unisson asynchrone auto-stabilisant tolérant aux fautes pour tout rayon de confinement si l'une des conditions suivantes est vérifiée :

1. Deux processeurs ou plus peuvent subir un crash.
2. Le démon est inéquitable.
3. Le démon est faiblement équitable et l'unisson est minimal ou prioritaire.
4. Le démon est fortement équitable, le degré du système réparti est supérieur à 3 et l'unisson est minimal ou prioritaire.

Au vu de ces résultats d'impossibilité, le chapitre 11 s'attache à étudier la possibilité des cas restants. Nous fournissons un protocole strictement stabilisant pour l'unisson asynchrone pour les systèmes répartis réduits à une chaîne ou à un anneau sous un démon fortement équitable. Ce protocole possède un rayon de confinement nul, ce qui est évidement optimal, est minimal et prioritaire, ce qui rend l'ensemble des hypothèses nécessaire d'après le chapitre 10. Enfin, nous prouvons que le temps de stabilisation de ce protocole strictement stabilisant est optimal.

Les résultats présentés dans le chapitre 10 ont fait l'objet d'une publication dans Theoretical Computer Science [DPBT11] et dans les actes du 23rd International Symposium on Distributed Computing (DISC 2009) [DPBT09]. Le chapitre 11 a conduit à une communication dans les actes de la 14th International Conference On Principles Of DIstributed Systems (OPODIS 2010) [DPBNT10].

A.3.3 Perspectives

Le chapitre 12 propose une généralisation du problème de l'unisson asynchrone dans lequel l'écart maximal autorisé entre les horloges de deux processeurs voisins est une constante arbitraire. Nous montrons alors que les résultats présentés dans cette partie sont aisément généralisables à ce problème.

Les principales questions encore ouvertes sur l'unisson asynchrone en présence de fautes transitoires et de fautes Byzantines intermittentes sont les suivantes :

1. Nous conjecturons que la minimalité et la priorité du protocole sont nécessaires à la stabilisation stricte déterministe. En revanche, la définition d'une stabilisation stricte probabiliste permettrait certainement de contourner les résultats d'impossibilité du chapitre 10.

2. Les résultats présentés dans cette partie sont valables dans le modèle à états (modèle à atomicité forte). L'extension de ces résultats à d'autres modèles pourrait être intéressante même si nous conjecturons qu'une atomicité plus fine tendrait à étendre les résultats d'impossibilité au cas de l'anneau.

A.4 Arbre couvrant

A.4.1 Contexte

Dans la quatrième partie de cette thèse, nous nous intéressons à un problème fondamental des systèmes répartis : la construction d'arbre couvrant [Gär03]. En effet, les systèmes répartis sont en partie caractérisés par le fait que les processeurs doivent communiquer pour résoudre des tâches non triviales. C'est pourquoi l'optimisation des communications est au cœur de nombreux protocoles répartis. Il existe plusieurs manières d'optimiser les communications dans un système réparti. Une des plus simple consiste à minimiser le nombre de liens de communications utilisés par les processeurs pour communiquer entre eux. Un arbre couvrant est en ce sens la meilleure structure de communication possible étant donné qu'il permet par définition à tous les processeurs de communiquer tout en minimisant le nombre de liens de communications réservés. Cette optimalité des arbres couvrants explique qu'il existe de nombreux protocoles répartis pour construire des arbres couvrants utilisés comme brique de base pour des applications réparties plus complexes nécessitant une optimisation des communications.

Pour un graphe de communication donné, il existe de nombreux arbres couvrants différents (par exemple, un graphe complet admet n^{n-2} arbres couvrants distincts). Même si tous ces arbres couvrants minimisent par définition le nombre de liens de communications réservés, ils ne sont pas tous équivalents. En effet, nous pouvons distinguer d'autres critères pour distinguer les arbres couvrants. Par exemple, un arbre couvrant en largeur [HC92] permet de minimiser le délai de communication entre un processeur distingué (la racine) et tout autre processeur tandis qu'un arbre de poids minimal [GHS83] minimise le coût global de l'arbre (dans le cas de graphes de communication valués).

Cette partie se concentre sur la construction d'arbre couvrant selon une métrique maximisable. Intuitivement, une métrique est une fonction permettant de calculer une distance le long de tout chemin du graphe de communication. Une métrique est maximisable [GS03] lorsqu'il existe toujours un arbre couvrant qui maximise la métrique de chaque processeur par rapport à la racine du système réparti. Par example, le plus court chemin ou le flot d'un chemin sont des métriques maximisables.

L'objectif de cette partie est l'étude de la construction d'arbre couvrant selon toute métrique maximisable dans un système réparti sujet à des fautes transitoires et à des fautes Byzantines intermittentes. La chapitre 13 donne plus de détails sur le contexte de cette partie.

A.4.2 Contributions

La contribution principale de cette partie est la définition de trois nouveaux concepts de confinement de fautes Byzantines en auto-stabilisation. Nous les présentons succintement dans la suite.

1. La stabilisation forte est une variante affaiblie de la stabilisation stricte. Plus précisèment, le rayon de confinement de la stabilisation stricte est relâché dans le temps étant donné que nous autorisons les processeurs en dehors du rayon de confinement à être perturbés un nombre fini de fois par les processeurs Byzantins après la convergence.
2. La stabilisation stricte topologiquement dépendante affaiblit également les contraintes imposées par la stabilisation stricte. Nous généralisons le rayon de confinement de la stabilisation stricte à une aire de confinement. Cette aire de confinement est simplement l'ensemble des processeurs (en fonction du graphe de communication) qui sont infiniment souvent perturbés par les processeurs Byzantins.
3. La stabilisation forte topologiquement dépendante combine les deux idées présentées ci-dessus.

Les définitions formelles de ces nouveaux concepts de tolérance aux fautes sont présentées dans le chapitre 13.

Dans ce contexte, les contributions de la quatrième partie de cette thèse sont les suivantes.

1. Le chapitre 14 propose une étude de deux cas particuliers de métriques maximisables. Dans un premier temps, il démontre l'existence d'un protocole fortement stabilisant pour la construction de l'arbre en profondeur avec un rayon de confinement nul (la stabilisation stricte étant impossible pour toute construction d'arbre). Ensuite, il se concentre sur la construction de l'arbre en largeur et démontre l'impossibilité de la stabilisation forte pour tout rayon de confinement. Nous nous intéressons alors à la stabilisation topologiquement dépendante et fournissons un protocole réparti exhibant l'aire de confinement optimale par rapport à la stabilisation stricte et forte topologiquement dépendante.
2. Le chapitre 15 s'intéresse à la construction d'arbre couvrant par rapport à toute métrique maximisable. La première contribution de ce chapitre est de proposer un protocole réparti pour cette tâche qui exhibe une aire de confinement optimale aussi bien pour la stabilisation stricte topologiquement dépendante que pour la stabilisation forte topologiquement dépendante. Enfin, nous caractérisons l'ensemble des métriques maximisables autorisant l'existence d'une solution fortement stabilisante, ce qui permet de généraliser les résultats du chapitre précédent.

Les résultats du chapitre 14 apparaissent dans un article de IEEE Transactions on Parallel and Distributed Systems [DMT11b], dans une communication dans les actes du 12th International Symposium on Stabilization, Safety, and Security of

Distributed Systems (SSS 2010) [DMT10c], des 12èmes Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications (Algotel 2010) [DMT10a], et des the 13èmes Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications (Algotel 2011) [DMT11a]. Les résultats du chapitre 15 apparaissent dans une communication dans les actes du 24th International Symposium on Distributed Computing (DISC 2010) [DMT10b] et du 25th International Symposium on Distributed Computing (DISC 2011) [DMT11c].

A.4.3 Perspectives

Le chapitre 16 propose plusieurs perspectives ouvertes par les notions introduites dans cette partie. En plus des perspectives immédiates liées aux propriétés des protocoles proposés comme l'amélioration de l'adversaire toléré ou l'extension à d'autres classes de métriques, il peut être intéressant de se pencher sur certaines questions plus générales comme les suivantes :

1. Tous les protocoles fortement stabilisants présentés dans cette partie nécessitent un démon fortement contraint (équité forte). Nous conjecturons que c'est le cas pour tout protocole fortement stabilisant mais nous ne pouvons pas produire de preuve de cette conjecture pour le moment.
2. Les concepts de stabilisation forte et de stabilisation topologiquement dépendantes présentés dans cette partie n'ont été appliqués qu'aux problèmes dits statiques (*i.e.* calcul de point fixe). L'extension de ces notions à des problèmes dynamiques (par exemple exclusion mutuelle ou circulation de jeton) est une question ouverte.

A.5 Conclusion

Dans cette section, nous discutons des questions ouvertes par les travaux menés dans cette thèse.

Nouveaux concepts de tolérance aux fautes Cette thèse se concentre sur les protocoles répartis auto-stabilisants qui sont de plus capable de confiner les effets d'une classe donnée de fautes permamentes ou intermittentes (qui sont supposées se produire après la fin des fautes transitoires). Une voie intéressante pour de futures recherches est l'étude de l'intérêt de la combinaison de variantes de l'auto-stabilisation avec de tels confinements. Par exemple, est-il possible de fournir des protocoles répartis instantanément stabilisant (*i.e.* satisfaisant toujours la spécification du problème quelle que soit la configuration initiale) capable de contenir les effets de fautes Byzantines intermittentes ?

En général, l'introduction de nouveaux concepts de tolérance aux fautes devrait permettre d'enrichir la palette de solutions disponibles étant donné que le concepteur de protocole réparti doit réussir à réaliser un compromis délicat entre :

1. la puissance du concept de tolérance aux fautes choisi par rapport à la tolérance ou au confinement de fautes,
2. la classe de problèmes admettant une solution selon ce concept de tolérance aux fautes,
3. le coût de ce concept de tolérance aux fautes sur son problème particulier,
4. la criticité de l'application, et
5. la fréquence d'occurence des fautes dans le système considéré.

Par exemple, il est inutile de garantir des propriétés de tolérance aux fautes très fortes pour un coût très élévé (en termes de ressources utilisées comme la mémoire ou la bande passante) pour un protocole réparti de routage dans un réseau ne subissant des fautes qu'une fois par an. En revanche, si des vies humaines dépendent du bon comportement de l'application (par exemple, application de contrôle de traffic aérien), de fortes propriétés de tolérance aux fautes sont souhaitables au détriment du coût en ressources.

Restriction du pouvoir des processeurs Byzantins Une autre manière de compléter les résultats présentés dans cette thèse est de restreindre le modèle de faute Byzantine utilisé. Par exemple, nous avons considéré uniquement des processeurs Byzantins avec une puissance infinie (pouvant exécuter un nombre infini d'actions incorrectes). Un autre modèle possible pour les fautes Byzantines pourrait considérer une puissance finie (chaque processeur Byzantin ne peut exécuter qu'un nombre fini d'actions incorrectes dans chaque exécution). Plusieurs question intéressantes sont soulevées par ce modèle. Existe-t-il un compromis entre le nombre d'actions incorrectes et le rayon ou l'aire de confinement ? Existe-t-il un compromis entre le nombre total de perturbations causées par les processeurs Byzantins et le nombre de processeurs Byzantins ? En d'autres mots, est-ce qu'un processeur Byzantin unique est-il plus efficace pour perturber le système réparti qu'une équipe de processeurs Byzantins considérant le même nombre d'actions incorrectes ?

Une fois encore, le but est de fournir une large palette de protocoles répartis avec des propriétés de tolérance aux fautes variées de manière à permettre à l'utilisateur du système réparti de choisir la solution la plus adaptée à son application et à ses contraintes propres. En effet, il est inutile de fournir une tolérance "parfaite"aux fautes Byzantines pour un coût élévé si le système réparti considéré subit seulement des omissions de messages (pour lesquelles il est peut être possible de fournir une solution adaptée pour un coût plus faible). La définition d'une taxonomie rigoureuse des fautes qui permettrait une comparaison objective du pouvoir de tous les concepts de tolérance aux fautes nous semble une perspective intéressante.

Approche probabiliste Dans cette thèse, nous avons choisi d'étudier le point de vue déterministe du confinement de fautes permanentes ou intermitentes en auto-stabilisation. Une autre solution pour contourner les nombreux résultats d'impossibilité présentés dans cette thèse est de garantir la stabilisation du système réparti seulement avec forte probabilité.

Comme en auto-stabilisation, nous pensons qu'il est possible de définir plusieurs concepts probabilistes de tolérance aux fautes permamentes ou intermittentes en auto-stabilisation. Par exemple, nous pourrions définir un concept de tolérance aux fautes qui assure des propriétés similaires à la stabilisation stricte avec une espérance bornée sur le temps de convergence (au lieu d'un temps de convergence borné) ou une clôture probabiliste. Une variante de la stabilisation forte pourrait être définie pour assurer une espérance bornée sur le nombre de perturbations. La question principale serait alors d'évaluer les bénéfices de l'approche probabiliste sur les rayons ou aires de confinement par rapport aux solutions déterministes. En d'autres mots, le but est toujous de permettre à l'utilisateur final du système réparti de trouver le meilleur compromis entre les propriétés de tolérance aux fautes, les difficultés d'implémentation, le coût des protocoles répartis et les résultats d'impossibilité.

Autres modèles de systèmes répartis Une intéressante possibilité d'extension des résultats présentés dans cette thèse est de considérer d'autres modèles de systèmes répartis. Nous pouvons présenter au moins deux voies de recherche :

1. Nous pouvons considérer des modèles incluant des nouvelles hypothèses très variées. De nombreux modèles de nouveaux systèmes répartis ont été décrit ces dernières années. Par exemple, nous pouvons considérer des systèmes répartis dynamiques dans lesquels les liens de communication peuvent apparaître et disparaître au fil du temps, des réseaux de capteurs dans lequels les processeurs communiquent par radio et disposent seulement d'une batterie limitée ou encore les réseaux de robots dans lesquels les processeurs sont équipés de capacités motrices. L'étude de la tolérance conjointe à plusieurs type de fautes dans de tels systèmes nous paraît être un réel défi.
2. Nous pouvons également nous intéresser à des modèles de systèmes répartis plus réalistes que ceux proposés dans cette thèse. Une question ouverte est la conception de transformateurs automatiques pour de tels modèles qui assurent la préservation des propriétés de tolérance aux fautes du protocole réparti original.

Index

A

Action 13
  Allowed 13
  Merged 21
  Projection of an 19
Activation 15
Atomic register simulation
  Specification 76

B

Bounded labeling system 94
  Stabilizing 96

C

Chain 12
Closure 59
Communication graph 11
  Anonymous 11
  Connected 12
  Identified 11
Communication subgraph 12
Complete communication graph 12
Configuration 13
Containment radius 65
Convergence 59
Correct vertex 23
Cycle 11
  Elementary 11

D

Daemon 14
  Boundedness 35
  Canonical 42
  Class 41
  Classical 50
  Distributed protocol running under a 14
  Distribution 30
  Enabledness 37
  Execution allowed by a 14
  Gouda fairness 33
  Minimal class 42
  More powerful relation 42
  Strong fairness 32
  Synchronous 48
  Transformer 51
  Weak fairness 32
Data-link communication 80
  Specification 82
Degree 12
Diameter 12
Distance 12
Distributed protocol 14
Distributed system 9
  Advantages 10
  Characteristics 10

E

Enabled vertex 15
Execution 14
Execution property 44
  Impossibility 44
  Satisfaction 44

F

Fault 18
  Byzantine 20
  Crash 20
  Execution subject to a 20
  Global 20
  Local 20
  Permanent 20
  Transient 20
Fault pattern 21
  Composite 25
  Execution subject to a 21
  Intermittent 25
  Permanent Byzantine 25

- Permanent crash . . . 24
- Transient . . . 24

Fault-containing self-stabilization . . . 62
Fault-tolerance . . . 57
- Masking . . . 57

M

Maximal Degree . . . 12
Maximum metric spanning tree construction
- Specification . . . 170

Member . . . 13
Message performance . . . 83
Metric . . . 166
- Assigned . . . 168
- Bounded . . . 169
- Fixed point . . . 199
- Maximizable . . . 168
- Monotonic . . . 169
- Strictly decreasing . . . 199
- Used value . . . 199

Model of computation
- Generic . . . 13
- Message passing . . . 16
- State . . . 15

N

New/old inversion . . . 74

O

Output variables . . . 172

P

Path . . . 11
- Elementary . . . 11
- Rooted . . . 168

Probabilistic stabilization . . . 60
Pseudo-stabilization . . . 60
- Fault-tolerant . . . 63

R

Register . . . 72
- Atomic . . . 74
- Regular . . . 73
- Safe . . . 73

Ring . . . 12
Robustness . . . 58

S

Self-stabilization . . . 58, 59
- Byzantine-tolerant . . . 64
- Fault-tolerant . . . 62, 125

Snap-stabilization . . . 61
Spanning forest . . . 12
Spanning tree . . . 12
- Maximum metric . . . 168

Specification . . . 58
Specification predicate . . . 172
Strict-stabilization . . . 65, 172
- Topology-aware . . . 176

Strong-stabilization . . . 174
- Topology-aware . . . 177

Super-stabilization . . . 61

T

Tree . . . 12

U

Unison
- Classical . . . 122
- Minimality . . . 123
- Priority . . . 124
- Specification . . . 123

W

Weak-stabilization . . . 61

List of Notations

α Action of g . . . 13

Γ Set of configurations of the system . . . 13

γ Configuration of the system . . . 13

$\gamma(e)$ State of edge e . . . 17

$\gamma(v)$ State of vertex v . . . 15

$\mathcal{D}$ Set of daemons . . . 14

μ Member of the system . . . 13

$\overleftarrow{uv}$ Communication channel for acknowledgements sent by v to u . . . 17

$\overrightarrow{uv}$ Communication channel for messages sent by u to v . . . 17

Π Set of distributed protocols on g . . . 14

π Distributed protocol on g . . . 14

σ Execution of π on g . . . 14

Σ_Π Set of executions of distributed protocols on g starting from configurations of Γ . . . 14

Σ_π Set of executions of π on g starting from configurations of Γ . . . 14

$deg(g)$ Maximal degree of communication graph g . . . 12

$deg(g, v)$ Degree of vertex v . . . 12

$diam(g)$ Diameter of communication graph g . . . 12

$dist(g, u, v)$ Distance between vertices u and v . . . 12

$\tilde{A}$ Set of actions of g . . . 13

A Set of allowed actions of g . . . 14

$Act(\alpha)$ Set of vertices activated during the action α . . . 15

d Daemon . . . 14

E Set of edges . . . 11

$Ena(\gamma, \pi)$ Set of enabled vertices by π in γ . . . 15

$g = (V, E)$ Communication graph . . . 11

M Set of member of the sysem . . . 13

m Number of edges . . . 12

n Number of vertices . . . 12

N_v Set of neighbors of vertex v . . . 12

V Set of vertices . . . 11

Bibliography

[AAD+10] Noga Alon, Hagit Attiya, Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Brief announcement: Sharing memory in a self-stabilizing manner. In *24th International Symposium on Distributed Computing (DISC 2010)*, 2010. 4, 77, 80, 246

[AAD+11] Noga Alon, Hagit Attiya, Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Pragmatic self-stabilization of atomic memory in message-passing systems. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*, 2011. 4, 77, 246

[AB93] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993. 80

[ABND90] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. In *9th Annual ACM Symposium on Principles of Distributed Computing (PODC 1990)*, pages 363–375, 1990. 76

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995. 76, 102, 104, 246

[Abr03] Uri Abraham. Self-stabilizing timestamps. *Theoretical Computer Science*, 308(1-3):449–515, 2003. 95

[ADGF+07] Emmanuelle Anceaume, Carole Delporte-Gallet, Hugues Fauconnier, Michel Hurfin, and Josef Widder. Clock synchronization in the byzantine-recovery failure model. In *11th International Conference on Principles of Distributed Systems (OPODIS 2007)*, pages 90–104, 2007. 18

[AH93] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures (extended abstract). In *7th International Workshop on Distributed Algorithms (WDAG 1993)*, pages 174–188, 1993. 62, 63, 244

[AKM+07] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE Transactions on Dependable and Secure Computing*, 4(3):180–190, 2007. 119

[AKY90] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self stabilizing protocols for general networks. In *4th International Workshop on Distributed Algorithms (WDAG 1990)*, pages 15–28, 1990. 163

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure

computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. 18

[Att10] Hagit Attiya. Robust simulation of shared memory: 20 years after. *The Distributed Computing Column of the Bulletin of the European Association for Theoretical Computer Science*, (100):99–113, 2010. 76

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing, Fundamentals, Simulations, and Advanced Topics*. John Wiley and Sons, 2004. 18

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Jourhnal of ACM*, 32(4):804–823, 1985. 119, 120

[BB04] Lélia Blin and Franck Butelle. The first approximated distributed algorithm for the minimum degree spanning tree problem on general graphs. *International Journal of Foundations of Computer Science*, 15(3):507–516, 2004. 164

[BCD95] Joffroy Beauquier, Stéphane Cordier, and Sylvie Delaët. Optimum probabilistic self-stabilization on uniform rings. In *2nd Workshop on Self-stabilizing Systems (WSS 1995)*, pages 15.1–15.15, 1995. 35

[BDH06] Joffroy Beauquier, Sylvie Delaët, and Sammy Haddad. Necessary and sufficient conditions for 1-adaptivity. In *20th IEEE International Conference on Paralle and Distributed Systems (IPDPS 2006)*, 2006. 62

[BDKM96] Joffroy Beauquier, Oliver Debas, and Synnöve Kekkonen-Moneta. Fault-tolerant and self-stabilizing ring orientation. In *3rd International Colloquium on Structural Information & Communication Complexity (SIROCCO 1996)*, pages 59–72, 1996. 63

[BDPBM02] Joffroy Beauquier, Ajoy Kumar Datta, Maria Potop-Butucaru, and Frédéric Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *Chicago Journal of Theoretical Computer Science*, 2002. 35, 53, 54

[BDPV07] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Snap-stabilization and pif in tree networks. *Distributed Computing*, 20(1):3–19, 2007. 61, 149

[Ber67] Claude Berge. *Théorie des graphes et ses applications*. Collection universitaire de mathématiques. Dunod, 1967. 11

[BGM93] James E. Burns, Mohamed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993. 60, 80, 175, 244

[BK07] Janna Burman and Shay Kutten. Time optimal asynchronous self-stabilizing spanning tree. In *21st International Symposium on Distributed Computing (DISC 2007)*, pages 92–107, 2007. 165

[BKM97a] Joffroy Beauquier and Synnöve Kekkonen-Moneta. Fault-tolerance and self stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International Journal of Systems Science*, 28(11):1177–1187, 1997. 63, 81

[BKM97b] Joffroy Beauquier and Synnöve Kekkonen-Moneta. On ftss-solvable distributed problems. In *3rd Workshop on Self-stabilizing Systems (WSS 1997)*, pages 64–79, 1997. 63

[BLB95] Franck Butelle, Christian Lavault, and Marc Bui. A uniform self-stabilizing minimum diameter tree algorithm (extended abstract). In *9th International Workshop on Distributed Algorithms (WDAG 1995)*, pages 257–272, 1995. 165

[Blo88] Bard Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Computers*, 37(12), 1988. 75

[BODH08] Michael Ben-Or, Danny Dolev, and Ezra N. Hoch. Fast self-stabilizing byzantine tolerant digital clock synchronization. In *27th ACM Symposium on Principles of Distributed Computing (PODC 2008)*, pages 385–394, 2008. 64, 121

[BP08] Christian Boulinier and Franck Petit. Self-stabilizing wavelets and ρ-hops coordination. In *22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS08)*, pages 1–8, 2008. 53, 54

[BPBJ01] Joffroy Beauquier, Maria Potop-Butucaru, and Colette Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In *5th International Workshop on Self-Stabilizing Systems (WSS 2001)*, pages 19–34, 2001. 35, 53, 55

[BPBJDL02] Joffroy Beauquier, Maria Potop-Butucaru, Colette Johnen, and Jérôme Olivier Durand-Lose. Token-based self-stabilizing uniform algorithms. *Journal of Parallel and Distributed Computing*, 62(5):899–921, 2002. 52, 53

[BPBR09] Lélia Blin, Maria Potop-Butucaru, and Stephane Rovedakis. A super-stabilizing log()-approximation algorithm for dynamic steiner trees. In *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*, pages 133–148, 2009. 165

[BPBR11] Lélia Blin, Maria Potop-Butucaru, and Stephane Rovedakis. Self-stabilizing minimum degree spanning tree within one from the optimal degree. *Journal of Parallel and Distributed Computing*, 71(3):438–449, 2011. 164, 165

[BPV04] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In *23rd ACM Symposium on Principles of Distributed Computing (PODC 2004)*, pages 150–159, 2004. 121, 124

[BPV05] Christian Boulinier, Franck Petit, and Vincent Villain. Synchronous vs. asynchronous unison. In *7th International Symposium on Self-Stabilizing Systems (SSS 2005)*, pages 18–32, 2005. 121, 124

[BPV06] Christian Boulinier, Franck Petit, and Vincent Villain. Toward a time-optimal odd phase clock unison in trees. In *8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*, pages 137–151, 2006. 121

[CD94] Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994. 165

[CDV09] Alain Cournier, Swan Dubois, and Vincent Villain. A snap-stabilizing point-to-point communication protocol in message-switched networks. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009)*, pages 1–11, 2009. 61

[CFG92] Jean-Michel Couvreur, Nissim Francez, and Mohamed G. Gouda. Asynchronous unison (extended abstract). In *12th International Conference on Distributed Computing Systems (ICDCS 1992)*, pages 486–493, 1992. 119, 120, 121, 122, 124, 247

[CHK93] Gen-Huey Chen, Michael E. Houle, and Ming-Ter Kuo. The steiner problem in distributed computing systems. *Information Sciences*, 74(1-2):73 – 96, 1993. 164

[CKW00] Soma Chaudhuri, Martha J. Kosa, and Jennifer L. Welch. One-write algorithms for multivalued regular and atomic registers. *Acta Informatica*, 37(3):161–192, 2000. 75

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transaction on Computer Systems*, 3(1):63–75, 1985. 119

[Cor01] T.H. Cormen. *Introduction to algorithms*. MIT electrical engineering and computer science series. MIT Press, 2001. 164

[CS96] Manhoi Choy and Ambuj K. Singh. Localizing failures in distributed synchronization. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):705–716, 1996. 66

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. 63

[DD05] Ariel Daliot and Danny Dolev. Self-stabilization of byzantine protocols. In *7th International Symposium on Self-Stabilizing Systems (SSS 2005)*, pages 48–67, 2005. 64

[DD06] Ariel Daliot and Danny Dolev. Self-stabilizing byzantine agreement. In *25th Annual ACM Symposium on Principles of Distributed Computing (PODC 2006)*, pages 143–152, 2006. 64

[DDNT10] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. Snap-stabilization in message-passing systems. *Journal of Parallel and Distributed Computing*, 70(12):1220–1230, 2010. 81

[DDPBT11a] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Communication optimalement stabilisante sur canaux non fiables et non fifo. In *13èmes Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications (Algotel 2011)*, 2011. 4, 77, 246

[DDPBT11b] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Stabilizing data-link over non-fifo channels with optimal fault-resilience. *Information Processing Letters*, 111(18):912–920, 2011. 4, 77, 246

[DDT06] Sylvie Delaët, Bertrand Ducourthial, and Sébastien Tixeuil. Self-stabilization with r-operators revisited. *Journal of Aerospace Computing, Information, and Communication*, 3(10):498–514, 2006. 166

[DGDF10] Carole Delporte-Gallet, Stéphane Devismes, and Hugues Fauconnier. Stabilizing leader election in partial synchronous systems with crash failures. *Journal of Parallel and Distributed Computing*, 70(1):45–58, 2010. 63, 244

[DGX11] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Fault containment in weakly stabilizing systems. *Theoretical Computer Science*, 412(33):4297–4311, 2011. 62

[DH97] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997, 1997. 61

[DH01] Shlomi Dolev and Ted Herman. Dijkstra's self-stabilizing algorithm in unsupportive environments. In *5th International Workshop on Self-Stabilizing Systems (WSS 2001)*, pages 67–81, 2001. 75

[DH07] Danny Dolev and Ezra N. Hoch. On self-stabilizing synchronous actions despite byzantine attacks. In *21st International Symposium on Distributed Computing (DISC 2007)*, pages 193–207, 2007. 64, 121

[DHT04] Philippe Duchon, Nicolas Hanusse, and Sébastien Tixeuil. Optimal randomized self-stabilizing mutual exclusion in synchronous rings. In *18th Symposium on Distributed Computing (DISC 2004)*, pages 216–229, 2004. 28, 48, 61

[Dij71] Edsger W. Dijkstra. A short introduction to the art of programming. circulated privately, 1971. 164

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of ACM*, 17(11):643–644, 1974. 13, 14, 16, 30, 31, 52, 53, 58, 59, 242, 243

[DIM90] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *9th Annual ACM Symposium on Principles of Distributed Computing (PODC 1990)*, pages 103–117, 1990. 165

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993. 13, 80, 163, 165

[DIM97a] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM Journal on Computing*, 26(1):273–290, 1997. 71, 76, 245

[DIM97b] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997. 80, 149

[DJPV00] Ajoy K. Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing*, 13:207–218, 2000. 53, 54, 165

[DKS10] Shlomi Dolev, Ronen I. Kat, and Elad Michael Schiller. When consensus meets self-stabilization. *Journal of Computer and System Sciences*, 76(8):884–900, 2010. 95

[DMT10a] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. Construction auto-stabilisante d'arbre couvrant en dépit d'actions malicieuses. In *12èmes Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications (Algotel 2010)*, 2010. 5, 171, 251

[DMT10b] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. The impact of topology on byzantine containment in stabilization. In *24th International Symposium on Distributed Computing (DISC 2010)*, 2010. 5, 171, 251

[DMT10c] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. On byzantine containment properties of the min+1 protocol. In *12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2010)*, 2010. 5, 171, 251

[DMT11a] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. Auto-stabilisation et confinement de fautes malicieuses : Optimalité du protocole min+1. In *13èmes Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications (Algotel 2011)*, 2011. 5, 171, 251

[DMT11b] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. Bounding the impact of unbounded attacks in stabilization. *IEEE Transactions on Parallel and Distributed Systems*, 2011. 5, 171, 250

[DMT11c] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. Maximum metric spanning tree made byzantine tolerant. In *25th Interna-*

tional Symposium on Distributed Computing (DISC 2011), 2011. 5, 171, 251

[DNT09] Praveen Danturi, Mikhail Nesterenko, and Sébastien Tixeuil. Self-stabilizing philosophers with generic conflicts. *ACM Transactions of Adaptive and Autonomous Systems*, 4(1), 2009. 30, 53, 54

[Dol97] Shlomi Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Real-Time Systems*, 12(1):95–107, 1997. 121

[Dol00] Shlomi Dolev. *Self-stabilization*. MIT Press, 2000. 18, 60, 80

[DPBNT10] Swan Dubois, Maria Potop-Butucaru, Mikhail Nesterenko, and Sébastien Tixeuil. Self-stabilizing byzantine asynchronous unison. In *14th International Conference On Principles Of DIstributed Systems (OPODIS 2010)*, 2010. 5, 32, 125, 248

[DPBT00] Ajoy K Datta, Maria Potop-Butucaru, and Sébastien Tixeuil. Self-stabilizing mutual exclusion using unfair distributed scheduler. In *14th IEEE International Parallel and Distributed Processing SYmposium (IPDPS 2000)*, pages 465–470, 2000. 30, 32, 35

[DPBT04] Ajoy K. Datta, Maria Potop-Butucaru, and Sébastien Tixeuil. Self-stabilizing mutual exclusion with arbitrary scheduler. *The Computer Journal*, 47(3):289–298, 2004. 32, 53, 54, 60

[DPBT09] Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Brief announcement: Dynamic FTSS in Asynchronous Systems: the Case of Unison. In *23rd International Symposium on Distributed Computing (DISC 2009)*, 2009. 5, 125, 248

[DPBT11] Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Dynamic ftss in asynchronous systems: the case of unison. *Theoretical Computer Science*, 412(29):3418–3439, 2011. 5, 32, 125, 248

[DPV11a] Stephane Devismes, Franck Petit, and Vincent Villain. Autour de l'auto-stabilisation. partie i : Techniques généralisant l'approche. *Technique et Science Informatiques*, 30(2489):1–22, 2011. 60

[DPV11b] Stephane Devismes, Franck Petit, and Vincent Villain. Autour de l'auto-stabilisation. partie ii : Techniques spécialisant l'approche. *Technique et Science Informatiques*, 30(2489):23–50, 2011. 60

[DS97] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, 1997. 94, 95

[DT01] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3):147–162, 2001. 166

[DT03] Bertrand Ducourthial and Sébastien Tixeuil. Self-stabilization with path algebra. *Theoretical Computer Science*, 293(1):219–236, 2003. 166

[DT06] Shlomi Dolev and Nir Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithms. In *10th International Conference on Principles of Distributed Systems (OPODIS 2006)*, pages 230–243, 2006. 80, 81

[DTY08] Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. Weak vs. self vs. probabilistic stabilization. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, June 2008. 35, 61, 62, 159

[DW95] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults (abstract). In *14th Annual ACM Symposium on Principles of Distributed Computing (PODC 1995)*, page 256, 1995. 63, 64, 244

[DW97] Shlomi Dolev and Jennifer L. Welch. Wait-free clock synchronization. *Algorithmica*, 18(4):486–511, 1997. 64, 121

[DW04] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004. 64, 121

[Erd63] Paul Erdös. On a problem in graph theory. *The Mathematical Gazette*, 47(361):220–223, 1963. 96

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, 1985. 27, 119, 247

[FMP06] Laurent Fribourg, Stéphane Messika, and Claudine Picaronny. Coupling and self-stabilization. *Distributed Computing*, 18(3):221–232, 2006. 28

[FR92] Martin Fürer and Balaji Raghavachari. Approximating the minimum degree spanning tree to within one from the optimal degree. In *3rd Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms (SODA 1992)*, pages 317–324, 1992. 164

[Gär03] Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report ic/2003/38, EPFL, 2003. 60, 165, 181, 249

[GGH+04] Martin Gairing, Wayne Goddard, Stephen T. Hedetniemi, Petter Kristiansen, and Alice A. McRae. Distance-two information in self-stabilizing algorithms. *Parallel Processing Letters*, 14(3-4):387–398, 2004. 53, 54

[GGHP96] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *15th ACM Symposium on Principles of Distributed Computing (PODC 1996)*, pages 45–54, 1996. 165

[GGHP07] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing distributed protocols. *Distributed Computing*, 20(1):53–73, 2007. 62

[GH90] Mohamed G. Gouda and Ted Herman. Stabilizing unison. *Information Processing Letters*, 35(4):171–175, 1990. 119, 120, 121, 122, 124, 247

[GH97] Mohamed G. Gouda and F. Furman Haddix. The linear alternator. In *3rd Workshop on Self-stabilizing Systems (WSS 1997)*, pages 31–47, 1997. 52, 53

[GH07] Mohamed G. Gouda and F. Furman Haddix. The alternator. *Distributed Computing*, 20(1):21–28, 2007. 52, 53

[GHJT08] Wayne Goddard, Stephen T. Hedetniemi, David Pokrass Jacobs, and Vilmar Trevisan. Distance- k knowledge in self-stabilizing algorithms. *Theoretical Computer Science*, 399(1-2):118–127, 2008. 53, 54

[GHS83] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1983. 164, 249

[GK10] Nabil Guellati and Hamamache Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70(4):406–415, 2010. 60

[GLS92] Rainer Gawlick, Nancy A. Lynch, and Nir Shavit. Concurrent timestamping made simple. In *International Symposium on Theory of Computing and Systems (ISTCS 1992)*, pages 171–183, 1992. 95

[GLS10] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010. 76

[GM91] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991. 80

[Gou01] Mohamed G. Gouda. The theory of weak stabilization. In *5th International Workshop on Self-Stabilizing Systems (WSS 2001)*, pages 114–123, 2001. 32, 61, 158

[GP93] Ajei S. Gopal and Kenneth J. Perry. Unifying self-stabilization and fault-tolerance (preliminary version). In *12th Annual ACM Symposium on Principles of Distributed Computing (PODC 1993)*, pages 195–206, 1993. 62, 63, 244

[GR06] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, 2006. 18

[GS71] R. L. Graham and J. H. Spencer. A constructive solution to a tournament problem. *Canadian Mathematical Bulletin*, 14(1):45–48, 1971. 98

[GS94] Mohamed G. Gouda and Marco Schneider. Maximum flow routing. In *Joint Conference on Information Sciences (JCIS 1994)*, 1994. 164

[GS99] Mohamed G. Gouda and Marco Schneider. Stabilization of maximal metric trees. In *ICDCS Workshop on Self-stabilizing Systems (WSS 1999)*, pages 10–17, 1999. 164, 165, 201, 202, 203

[GS03] Mohamed G. Gouda and Marco Schneider. Maximizable routing metrics. *IEEE/ACM Transactions on Networks*, 11(4):663–675, 2003. 164, 165, 166, 168, 169, 249

[HC92] Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2):109–117, 1992. 163, 165, 185, 187, 249

[HDD06] Ezra N. Hoch, Danny Dolev, and Ariel Daliot. Self-stabilizing byzantine digital clock synchronization. In *8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*, pages 350–362, 2006. 64, 121

[Her90] Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990. 27, 48, 60

[Her02] Ted Herman. A comprehensive bibliography on self-stabilization. http://www.cs.uiowa.edu/ftp/selfstab/bibliography/, 2002. 59

[HG95] Ted Herman and Sukumar Ghosh. Stabilizing phase-clocks. *Information Processing Letters*, 54(5):259–265, 1995. 121

[HL98] Shing-Tsaan Huang and Tzong-Jye Liu. Four-state stabilizing phase clock for unidirectional rings of odd size. *Information Processing Letters*, 65(6):325–329, 1998. 121

[HL02] Tetz C. Huang and Ji-Cherng Lin. A self-stabilizing algorithm for the shortest path problem in a distributed system. *Computers and Mathematics with Applications*, 43(1-2):103 – 109, 2002. 165

[HLCW10] Tetz C. Huang, Ji-Cherng Lin, Chih-Yuan Chen, and Cheng-Pin Wang. The worst-case stabilization time of a self-stabilizing algorithm under the weakly fair daemon model. *International Journal of Artificial Life Research*, 1(3):45–52, 2010. 32

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 522–529, 2003. 124

[HNM99] Rodney R. Howell, Mikhail Nesterenko, and Masaaki Mizuno. Finite-state self-stabilizing protocols in message-passing systems. In *ICDCS*

Workshop on Self-stabilizing Systems (WSS 1999), pages 62–69, 1999. 80

[HPT02] Jaap-Henk Hoepman, Marina Papatriantafilou, and Philippas Tsigas. Self-stabilization of wait-free shared memory objects. *Journal of Parallel and Distributed Computing*, 62(5):818–842, 2002. 75

[HV95] Sibsankar Haldar and K. Vidyasankar. Constructing 1-writer multi-reader multivalued atomic variable from regular variables. *Journal of the ACM*, 42(1):186–203, 1995. 75

[HW97] Shing-Tsaan Huang and Lih-Chyau Wuu. Self-stabilizing token circulation in uniform networks. *Distributed Computing*, 10(4):181–187, 1997. 165

[IL93] Amos Israeli and Ming Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, 1993. xvi, 76, 77, 93, 94, 95, 96, 97, 102, 111

[IS92] Amos Israeli and Amnon Shaham. Optimal multi-writer multi-reader atomic register. In *11th Annual ACM Symposium on Principles of Distributed Computing (PODC 1992)*, pages 71–82, 1992. 75

[JADT02] Colette Johnen, Luc Alima, Ajoy K. Datta, and Sébastien Tixeuil. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters*, 12(3-4):327–340, 2002. 52, 53

[JH09] Colette Johnen and Lisa Higham. Fault-tolerant implementations of regular registers by safe registers with applications to networks. In *10th International Conference on Distributed Computing and Networking (ICDCN 2009)*, pages 337–348, 2009. 75

[JT03] Colette Johnen and Sébastien Tixeuil. Route preserving stabilization. In *6th International Symposium on Self-Stabilizing Systems (SSS 2003)*, pages 184–198, 2003. 61

[KA98] Sandeep S. Kulkarni and Anish Arora. Multitolerance in distributed reset. *Chicago Journal of Theoritical Computer Science*, 1998, 1998. 19

[KA06] Sandeep S. Kulkarni and Mahesh Arumugam. Transformations for write-all-with-collision model. *Computer Communications*, 29(2):183–199, 2006. 71, 245

[Kar01] Mehmet Hakan Karaata. Self-stabilizing strong fairness under weak fairness. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):337–345, 2001. 32, 53, 55

[Kar05] Mehmet Hakan Karaata. An optimal self-stabilizing strarvation-free alternator. *Journal of Computer and System Sciences*, 71(4):480–494, 2005. 53, 55

[KC98] Mehmet Hakan Karaata and Pranay Chaudhuri. A self-stabilizing algorithm for strong fairness. *Computing*, 60(3):217–228, 1998. 32

[Kru56] Jr. Kruskal, Joseph B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956. 164

[KY97] Hirotsugu Kakugawa and Masafumi Yamashita. Uniform and self-stabilizing token rings allowing unfair daemon. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):154–162, 1997. 32

[KY02] Hirotsugu Kakugawa and Masafumi Yamashita. Uniform and self-stabilizing fair mutual exclusion on unidirectional rings under unfair distributed daemon. *Journal of Parallel and Distributed Computing*, 62(5):885–898, 2002. 30, 32

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communication of the ACM*, 17(8):453–455, 1974. 94

[Lam86a] Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986. 72, 73, 74, 245

[Lam86b] Leslie Lamport. On interprocess communication. part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986. 72, 73, 74, 75, 245

[LH01] Tzong-Jye Liu and Shing-Tsaan Huang. Phase synchronization on asynchronous uniform rings with odd size. *IEEE Transactions on Parallel and Distributed Syst.ems*, 12(6):638–652, 2001. 121

[LMSM09] Sergey Legtchenko, Sébastien Monnet, Pierre Sens, and Gilles Muller. Churn-resilient replication strategy for peer-to-peer distributed hash-tables. In *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*, pages 485–499, 2009. 11

[LS95] Chengdian Lin and Janos Simon. Possibility and impossibility results for self-stabilizing binary clocks on synchronous rings. In *2nd International Workshop on Self-Stabilizing Systems (WSS 1995)*, 1995. 121

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996. 9, 16, 18, 71, 72, 74, 81, 82, 245

[Mis91] Jayadev Misra. Phase synchronization. *Information Processing Letters*, 38(2):101–105, 1991. 120, 121, 247

[MR98] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998. 76

[MT07] Toshimitsu Masuzawa and Sébastien Tixeuil. Stabilizing link-coloration of arbitrary networks with unbounded byzantine faults. *International Journal of Principles and Applications of Information Science and Technology*, 1(1):1–13, 2007. 66

[NA02] Mikhail Nesterenko and Anish Arora. Tolerance to unbounded byzantine faults. In *21st Symposium on Reliable Distributed Systems (SRDS 2002)*, pages 22–29. IEEE Computer Society, 2002. 64, 65, 66, 172, 225, 227, 229, 244

[Ora01] Andy Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly Media, 2001. 11

[PBT00] Maria Potop-Butucaru and Sébastien Tixeuil. Self-stabilizing vertex coloring of arbitrary graphs. In *4th International Conference on Principles of Distributed Systems (OPODIS 2000)*, pages 55–70, 2000. 28, 48

[PBT07] Maria Potop-Butucaru and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007)*, page 46, 2007. 53, 54, 55

[Pri57] Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technology Journal*, 36:1389–1401, 1957. 164

[PT97] Marina Papatriantafilou and Philippas Tsigas. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters*, 7(3):321–328, 1997. 64, 121, 124

[PV07] Franck Petit and Vincent Villain. Optimal snap-stabilizing depth-first token circulation in tree networks. *Journal of Parallel and Distributed Computing*, 67(1):1–12, 2007. 61

[Ray00] Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Distributed Systems*. Morgan & Claypool Publishers, 2000. 18

[Ray10] Michel Raynal. *Fault-Tolerant Agreement in Synchronous Distributed Systems*. Morgan & Claypool Publishers, 2010. 18

[RH90] Michel Raynal and Jean-Michel Hélary. *Synchronization and control of distributed systems and programs*. Wiley series in parallel computing. Wiley, 1990. 120

[Rov09] Stephane Rovedakis. *Algorithmes auto-stabilisants de constructions d'arbres couvrants*. PhD thesis, Université d'Evry-Val-d-Essone, 2009. 165

[Sch97] Marco Schneider. *Flow Routing in Computer Networks*. PhD thesis, University of Texas at Austin, 1997. 165

[SOM05] Yusuke Sakurai, Fukuhito Ooshita, and Toshimitsu Masuzawa. A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In *8th International Conference on Principles of Distributed Systems (OPODIS 2005)*, pages 283–298, 2005. 66

[SS65] Esther Szekeres and George Szekeres. On a problem of schütte and erdös. *The Mathematical Gazette*, 49(369):290–293, 1965. 98

[Tel10] Gerard Tel. *Introduction to distributed algorithms*. Cambridge University Press, 2010. 9, 13, 18, 59

[TH94] Ming-Shin Tsai and Shing-Tsaan Huang. A self-stabilizing algorith for the shortest paths problem with a fully distributed demon. *Parallel Processing Letters*, 4:65–72, 1994. 164

[Tix09] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009. 18, 57, 60, 244

[Tol37] John Ronald Reuel Tolkien. *The Hobbit, or There and Back Again.* 1937. 1, 239

[Tol54a] John Ronald Reuel Tolkien. *The Fellowship of the Ring*. 1954. 1, 239

[Tol54b] John Ronald Reuel Tolkien. *The Two Towers*. 1954. 1, 239

[Tol55] John Ronald Reuel Tolkien. *The Return of the King*. 1955. 1, 239

[VA86] Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware (detailed abstract). In *27th Annual Symposium on Foundations of Computer Science (FOCS 1986)*, pages 233–243, 1986. 94

[Var00] George Varghese. Self-stabilization by counter flushing. *SIAM Journal on Computing*, 30(2):486–510, 2000. 80

Tolérer les fautes transitoires, permanentes et intermittentes

Résumé : Un système réparti est un système constitué d'un ensemble d'unités de calcul autonomes dotées de capacités de communication afin de résoudre une tâche globale. Ce modèle est suffisament général pour décrire tout type de réseau physique (réseau local, réseau de capteurs, ...). Lorsque la taille d'un système réparti devient importante ou lorsque ce système est déployé dans un environnement non contrôlé, la probabilité que certains éléments du système subissent des fautes (panne, corruption de mémoire, piratage, ...) devient non négligeable. Ces fautes peuvent être classifiées en fonction de leur durée, de leur étendue et de leur nature. Dans cette thèse, nous nous intéressons aux systèmes répartis capables de tolérer simultanément plusieurs types de fautes à travers l'étude de trois problèmes fondamentaux. Nous présentons ainsi un protocole réparti simulant un registre atomique mono-écrivan multi-lecteurs en présence de fautes transitoires et de fautes permanentes de type crash. Ce protocole repose sur deux outils ré-utilisables : un protocole de communication et un système d'estampillage borné. Ensuite, nous proposons une étude de la synchronisation faible d'horloges logiques en présence de fautes transitoires et de fautes intermittentes Byzantines. Nous prouvons de nombreux résultats d'impossibilité et nous fournissons un protocole optimal dans les cas non couverts par ces résultats. Finalement, nous définissons trois nouveaux concepts de tolérance pour les systèmes répartis sujets à des fautes transitoires et des fautes intermittentes Byzantines. Nous donnons un protocole de construction d'une vaste classe d'arbres couvrants optimal selon ces trois concepts.

Mots clés : système réparti, tolérance aux fautes, auto-stabilisation et confinement de fautes permanentes/intermittentes, registre atomique, unisson, arbre couvrant

Tolerating Transient, Permanent, and Intermittent Failures

Abstract: A distributed system is a system composed of a set of autonomous computation units endowed with communication abilities in order to solve a global task. This model is general enough to describe any kind of network (LAN, sensor network, ...). When the size of a distributed system gets larger or when it is deployed in hazardous environments, the possibility that some elements of the system are subject to faults (failure, memory corruption, hacking, ...) become impossible to elude. Faults can be classified according to duration, span, or nature. In this thesis, we focus on distributed systems that simultaneously tolerate several kinds of faults using three classical problems as case studies. We present first a distributed protocol simulating a single-writer multi-reader atomic register in the presence of transient faults and of permanent crash faults. This protocol relies on two re-usable tools: a communication primitive and a bounded timestamp scheme. Then, we study logical clock weak synchronization in the presence of transient faults and of intermittent Byzantine faults. We prove several impossibility results and provide a protocol that is optimal both with respect to impossibility result and with respect to recovery time. Finally, we define three new fault tolerance schemes in distributed systems that are subject to transient faults and to intermittent Byzantine faults. We design a protocol constructing a wide class of spanning trees that is optimal with respect to fault tolerance metrics defined for these three schemes.

Keywords: distributed system, fault tolerance, self-stabilization and containment of permanent/intermittent faults, atomic register, unison, spanning tree