N° d'ordre : 3383

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Brahim HAMID**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Distributed Fault-Tolerance Techniques for Local Computations

---

Soutenue le : 14 Juin 2007

Après avis des rapporteurs :

Vincent VILLAIN ...	Professeur
Stefan GRUNER	Professeur

Devant la commission d'examen composée de :

Vincent VILLAIN ...	Professeur	Président
Khalil DRIRA	HDR Chargé de Recherche CNRS	Rapporteur
Yves METIVIER	Professeur	Directeur de Thèse
Mohamed MOSBAH	Professeur	Directeur de Thèse

– 2007 –

Abstract

This thesis describes theoretical and practical contributions to study fault-tolerance in distributed computing systems, and to investigate the impact of graph structures to deal with fault-tolerance. Specifically, we deal with process failures and we consider transient (or temporary) and crash (or permanent) failures. Our purpose is to study such systems in the context of locality in the sense that the only permitted actions are those using local information. We use both local computations and message passing models. We build on two main techniques in fault-tolerance research: Failure detection and self-stabilization.

For faults that are bounded in time, they lead a system to reach an arbitrary state. We have proposed an extension of the local computations model to study self-stabilization as fault-tolerance against such faults. That is, we deal with local detection and correction of the not "legal" (or abnormal) configurations resulting from the arbitrary state. The problem of resolving conflict is studied in more detail and its stabilization time in terms of the needed synchronizations is given. For crash failures, we first propose a module to implement a failure detector in the local computations model. It is a module that gives its process its mind (possibly incorrect), whether another process may have crashed or not. By combining these techniques and having failure locality purpose in mind, we construct a novel framework to transform an intolerant algorithm to a fault-tolerant one taking benefit of the proofs of the the first one.

On the other hand, we augment the Visidia software to simulate failures and then fault-tolerance. The transient failures are simulated simply through views allowing to change the states of the nodes. To deal with crash failures, first we integrate the failure detector module to Visidia with an interface to measure its performances in order to reach expected behaviors. Second, through views, the user may stop the work of some node to simulate the crash. To circumvent the impossibility implementation, the system of the failure detector is distinguished from the algorithm one. That is, the failure detector is implemented with its required synchrony while the algorithm is still in asynchronous mode.

At the end, we use graph theory to help fault-tolerance following three aspects. We present a new formalization of the vertex connectivity using the notion of succorer sons. Then we propose a local distributed algorithm to test the 2-connectivity of graphs using a constructive approach in the two used models. These results illuminate how to build a bridge between these models. Second, extending the notion of succorer sons, we studied the maintenance of a forest of spanning trees in spite of crash failures. Another aspect that has been considered is efficiency taking benefit of the structural knowledge. Thus, we propose to use free triangle graphs to solve the problem of resolving conflicts in the presence of transient failures. Further, we study the maintenance of a spanning tree for k-connected graphs in spite of $k-1$ consecutive failures. In this case, our formalization is an occurrence of the Menger's theorem.

Résumé

Ce travail constitue une contribution théorique et pratique à l'étude de la tolérance aux pannes dans les système distribués, et de spécifier l'impacte de la théorie des graphes dans ce domaine. Précisément, on s'intéresse aux pannes de processus et nous considérons les pannes transitoires et les pannes franches. Notre but est d'étudier ces systèmes dans le contexte de la localité: Les seuls calculs autorisés sont ceux à base d'informations locales. Comme modèle, on utilisera les calculs locaux et à passage de messages. Nos constructions sont basées sur les techniques les plus répondues dans ce domaine: La détection de pannes et l'auto-stabilisation.

Pour les pannes de durée limitée, elles mènent le système dans un état arbitraire. Nous proposons une extension au modèle des calculs locaux pour exprimer l'auto-stabilisation. Ceci revient à capturer la notion de détection et de correction des configurations "illégales" qui résultent de l'état arbitraire du système. Le problème de la résolution de conflits est détaillé, et son temps de stabilisation est calculé en nombre de synchronisations. Pour les pannes franches, en premier nous proposons un module pour implanter un détecteur de pannes distribué pour les calculs locaux. C'est un module qui donne son avis à son processus (qui peut être incorrect), si un autre processus est en panne ou non. En combinant ces deux techniques et en gardant la localité comme but ultime, nous construisons un nouvel outil pour transformer un algorithme intolérant en un autre algorithme équivalent mais qui est tolérant aux pannes, dont la preuve de correction est déduite du premier.

D'une autre part, nous avons augmenté la plate-forme Visidia pour simuler les pannes. Les pannes transitoires sont simplement simulées à travers des vues permettant de changer l'état des noeuds. Pour les pannes franches, en premier le détecteur de pannes est intégré dans Visidia, en plus d'une interface pour mesurer ses performances pour atteindre les comportements attendus. En second, à travers des vues, l'utilisateur peut stopper le travail d'un noeud et simuler la panne d'un processus. Pour éviter les résultats d'impossibilité, le système de communication sous-jacent aux détecteurs est distingué de celui de l'algorithme: Le détecteur est implanté avec ses besoins de synchronisme tandis que l'algorithme reste asynchrone.

Pour terminer cette thèse, nous sollicitons la théorie de graphe pour aider la tolérance aux pannes selon trois aspects. Nous présentons une nouvelle formalisation du test de la connexité de graphes en utilisant la notion de fils de secours. Un algorithme distribué et local pour tester la 2-connexité est proposé dans les deux modèles. Ce résultat nous illumine à propos de la manière dont un pont peut être construit entre ces deux modèles. En second, le calcul des fils de secours est entendu pour étudier la maintenance de forêts d'arbres recouvrants en présence de pannes franches. Le dernier aspect est l'efficacité en tenant compte des connaissances structurelles. Les graphes sans triangles sont adaptés à la résolution des conflits et la maintenance présente un intérêt pour les graphes k-connexes en présence de $k-1$ pannes consécutives. Dans ce cas, notre formalisation devient une occurrence du théorème de Menger.

Acknowledgments

The pursuit of happiness may start with a thesis? A good and unanswered question.

I would like to thank my committee members Vincent VILLAIN, Khalil DRIRA, Stefan GRUNER, Yves METIVIER, and Mohamed MOSBAH for their friendship and wisdom.

Reaching back further to my postgraduate experiences at LaBRI (University of Bordeaux1), I'd like to thank my formers in Master at the university of Amiens.

I am grateful to my adviser, Prof Mohamed MOSBAH. He inspired my interest in distributed computing and supported me in many ways during the becoming of this thesis. I thank my second assessor, Prof Yves METIVIER he helped me a lot with his detailed and precious comments.

Many thanks go to Bertrand LE SAEC and Akka ZEMMARI. The discussions with them were always enlightening and I enjoyed to joint work with them a lot. Part of this thesis originates also from this joint work. I like to thank Prof. Vincent VILLAIN and Prof. Stefan GRUNER for proofreading and comments on parts of this thesis.

Thanks also to colleagues for their characteristically sage comments, and for guidance in the ways of leadership. I am indebted to all the members of the Visidia and Distributed algorithm GT. It has been a privilege to enjoy your warm humor and insightful criticism alike.

Finally, for the love of my family, both old and new, near and far. I am grateful to my family who provided the right environment for my studies and my work. My brother and my sisters and persons who are near to my heart even though we are always far away. To my friends for making me one of the family. Thanks go also to all other people that contributed in any way to the success of this thesis. To my circle of peers I offer special thanks...

Preface

In our society, distributed computing interests all information systems. The algorithms designed and implemented for special network topology or explicit environment may be easier to design, and often more efficient, but less flexible regardless of the network topology changes or environment evolution.

The most goal of our research is to deal with fault-tolerance in distributed environment. This work unifies four fundamental aspects in this area:

- Models
- Algorithms
- Experimentation
- Network structure

We start to show the motivations of our research about fault-tolerance in distributed computing systems. Then, we present well-known terminology and some relevant techniques.

The formal frameworks including tools to model networks, to formalize distributed computing and to encode distributed algorithms are presented in Chapter 1. Detailed systems for all following chapters are given. Since the system assumptions in these chapters differ, the overall system model is chosen such that it covers all these system models. Specific assumptions are presented only as needed. In fact, some models are presented in their formal state and other using informal descriptions. An overview of two examples of distributed computation of a spanning tree are also given.

Chapter 2 deals with the self-stabilization using relabeling systems. The notion of illegal configuration is introduced in the local computations model. Such an extension is used to encode states that disturb the functioning of applications. Automatic generation method of self-stabilizing distributed algorithms is proposed. The approach is illustrated with examples including the spanning tree computation and distributed resource allocation. For the last one, we propose an implementation and analysis based on the use of randomized synchronizations. Then, we study some graph structure for which this algorithm is executed optimally.

Chapter 3 focuses on the local failure detection services. We present and analyze our failure detector protocol based on local computations. Some experimentations are proposed to validate the power and analysis of such a protocol. A new view of failure detection measures is also discussed.

Chapter 4 is devoted to the formal design of local fault-tolerant distributed applications. Given a network which is improved using unreliable failure detectors, we propose a method

to design fault-tolerant distributed algorithms by means of local computations. The procedure is illustrated through examples of distributed spanning tree computations.

A tool to help the designers of fault-tolerant distributed applications is presented in Chapter 5. That is, a software to prototype applications based on distributed algorithms encoded in both local computations and message passing models. This part unifies the formal framework of fault-tolerant distributed algorithms, the use of failure detectors on Visidia platform. Example of simulation of fault-tolerant distributed algorithm shows the advantages of this approach. Then, we discuss how our platform can be used to prototype fault-tolerant distributed applications.

As the structure of the network and especially its connectivity is an important feature on the study of the fault-tolerance in distributed systems, we propose in Chapter 6 a protocol to test the 2-vertex connectivity of graphs in distributed setting. That is, a general and a constructive approach. The protocol is encoded in the local computations model and implemented in the message passing model using a set of procedures.

In Chapter 7, we use the same scheme of the previous algorithm to deal with the maintenance of a forest of spanning trees of a network in the presence of crash failures. A new formalization is given and an incremental algorithm is proposed. Then, we show the impact of the graph structure to design efficient solutions of problems in distributed environments in spite of failures. So we adapt our protocol to deal with the maintenance of a spanning tree in the presence of crash failures for k-connected graphs. That is a useful structure to design reliable applications for unreliable networks.

Some of the works presented in this thesis have been published separately as referred conference and journal papers:

Section 2.5 was published as: B. Hamid and M. Mosbah, An automatic approach to self-stabilization, Proceedings SNPD05, 6th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/ Distributed Computing, IEEE Computer Society, 2005, pp 123-128.

Section 2.6.2 was appeared as: B. Hamid and M. Mosbah, A Local Self-stabilizing Enumeration Algorithm, Proceedings DAIS06, Distributed Applications and Interoperable Systems, 6th IFIP WG 6.1 International Conference, LNCS 4025, Springer-Verlag, 2006, pp. 289-302. An extended version of this work is also published as: B. Hamid and M. Mosbah, A Local Enumeration Protocol in Spite of Corrupted Data, Journal Of Computer(JCP), Volume 1(7), Academy Publisher, 2006, pp. 9-20.

Section 3.4 was published as: B. Hamid and M. Mosbah, An implementation of a failure detector for local computations in graphs, Proceedings of the 23rd IASTED International multi-conference on parallel and distributed computing and networks, ACTA Press, 2005, pp 473-478.

Section 4.4 appeared as: B. Hamid and M. Mosbah, A formal model for fault-tolerance in distributed systems, Proceedings SAFECOM05, 24th International Conference on Computer Safety, Reliability and Security, LNCS 3688, Springer-Verlag, 2005, pp. 108-121.

Section 5.2 was published as: B. Hamid and M. Mosbah, Visualization of self-stabilizing distributed algorithms. Proceedings IV05, 9th International conference on information visualization, IEEE Computer Society, 2005, pp. 550-555. An extended version is accepted to be published as: B. Hamid and M. Mosbah, Self-stabilizing applications in distributed environments with transient faults, International Journal of Applied Mathematics and Computer Science.

The result of Section 5.3 appeared as: B. Hamid, A tool to design and prototype fault-tolerant distributed algorithms, Proceedings NOTERE06, New Technologies for Distributed Systems, 2006, HERMES, pp 311-322.

The first version of Section 6.3 is accepted for inclusion in the proceeding of the PDCS 2007 and in the IJECS Journal as: Distributed 2-Vertex Connectivity Test of Graphs Using Local Knowledge, with B. Le Saec and M. Mosbah.

The new version of this section is accepted for inclusion in the Proceedings ISPA 2007,The Fifth International Symposium on Parallel and Distributed Processing and Applications , LNCS , Springer-Verlag as: Distributed Local 2-Connectivity Test of Graphs and Applications, with B. Le Saec and M. Mosbah.

A partial result of Chapter 7 was presented in the Workshop on Graph Computation Models(GCM06) as: B. Hamid, Distributed Maintenance of Spanning Trees Using Graph Relabeling Systems, 2006. The result of Section 7.8.3, related to the message passing model, is accepted for publication at the VECOS 2007 proceeding as: Simple Distributed Algorithm for the Maintenance of a Spanning Tree, with B. Le Saec and M. Mosbah.

Some of the results in this thesis that are not yet been published are:

- Section 2.7 is in preparation as: Self-stabilizing Distributed Algorithm For Resolving Conflicts, with M. Mosbah and A. Zemmari.
- An extended version of the result of Section 6.3 is invited for possible publication in one of WASET's journals.
- The result of Section 7.8.3 related to the local computations model is in preparation as: Distributed Maintenance of a Spanning Tree of k-Connected Graphs, with B. Le Saec and M. Mosbah.

List of Figures

List of algorithms and procedures

Contents

Introduction

A distributed system is a system which involves several computers, processors or processes which cooperate in some way to do some task. However, such systems raise several issues not found in single processor systems. The main difficulty arises when faults occur. Faults may be hardware defects (link failures, crashes) but also they can refer to (software) errors which prevent a system to continue functioning in a correct manner. While in the past, it has been considered acceptable for services to be unavailable because of failures, this is rapidly changing because our society becomes more dependent on computer technology, and therefore the requirement for high reliability and availability for systems is continually increasing. The rising popularity of distributed systems and their applications in many domains (such as finance, booking, telecommunication) have increased the importance of developing methods to both detect and handle faults.

There are two principal approaches to improve the reliability of a system. The first is called *fault prevention* [Lap92] and the second approach is *fault-tolerance* [Sch90, AG93, AAE04]. The aim of these approaches is to provide a service in spite of the presence of faults in the system. Several research works refer to the paradigm of *fault-tolerance* as the ability of a system to recover, in a finite time, to reach a desired computation. Most existing fault-tolerance implementations in the literature propose global solutions which require to involve the entire system. As networks grow fast, detecting and correcting errors globally is no longer feasible. The solutions that deal with local detection and correction are rather essential because they are scalable and can be deployed even for large and evolving networks. Note, however, that in this case it is useful to have the correct (non faulty) parts of the network operating normally while recovering locally the faulty components.

Fault-tolerance and dependable systems research covers a wide spectrum of applications ranging across embedded real/time systems, commercial transaction systems, transportation systems, and military/space systems. The supporting research includes *system architecture*, *design techniques*, coding theory, *testing*, validation, *proof of correctness*, *modeling*, software reliability, operating systems, *parallel processing* and real/time processing. These areas often involve widely diverse core expertise including *formal logic*, *mathematics*, *stochastic modeling*, *graph theory*, hardware design and *software engineering*.

Usually, a distributed system is composed of two types of components: Processors (or machines) and communication channels between the processors. Distributed computing studies the computational activities performed on these systems. Such systems are subject to some paradigms including coordination due to the absence of centralization, and to much uncertainly result of failures: Processors may crash, information (knowledge) may be corrupted, lost or duplicated during the transmission and son forth. Fault-tolerant computing deals with

the study of reliable distributed systems that tolerate such uncertainly. Informally, a fault-tolerant algorithm ensures that after any failure, it converges from a faulty state to a correct one. It is equivalent to say that the system will automatically recover to reach a correct state in a finite time. Therefore, an algorithm is called fault-tolerant if it eventually starts to behave correctly regardless of the configuration with fault components.

When we study distributed computing, often we use models to denote some abstract representation (environment) of a distributed system. A distributed system (or network) is represented as a connected, undirected graph G denoted by (V, E). A node in V represents a process and an edge in E represents a bidirectional communication link. To encode distributed algorithms in such systems, we studied message passing model [Lyn96, AW98, Tel00] and local computations model [LMS99, BMMS02]. In the former model, the nodes have only local vision of the graph and communicate only with their neighbors by messages. The program executed at each node consists of a set of variables (state) and a finite set of actions. A node can write to its own variables, send and/or receive messages from its neighbors. In the latter model, the local state of a process (resp. link) is encoded by some labels attached to the corresponding vertex (resp. edge). A distributed algorithm is therefore encoded by means of local rewritings. Moreover, in both models, such algorithms can be implemented in a unified way on a platform called *Visidia* [BMMS01]. However, it has been assumed that components of such a system do not fail.

Because the paradigm of designing fault-tolerant distributed algorithms is challenging and exciting, we are interested in studying and designing fault-tolerant algorithms in our framework: Local computations model [LMS99]. The motivations of this thesis are on the one hand to formulate the properties of fault-tolerant algorithms by using those of rewriting systems yielding simple and formal proofs. On the other hand, as locality is an important feature of distributed computing, it is important to understand how to carry on local computations in the presence of faults. In this thesis, we present a modular approach based on graph transformations to deal with faults within the framework of local computations. Then, we show the possible implementation of such solutions in the message passing model. For efficiency reasons, we study the impact of some graph structure to our methodology, and then we take advantage to improve our solutions. Most of the presented protocols are implemented and tested on the Visidia software, requiring in some cases extension of the existing platform.

Before we embark on discussing any aspect of fault-tolerant systems, we have to define and sometimes to recall what we mean by a system, error, fault, failure and fault-tolerance. These terms have been used in a variety of ways in different contexts and terms fault, failure and error have often been used interchangeably. Here we adopt the same use of such terms. We start this thesis by a system presentation to describe the context of our study, to give some terminology and techniques. The remainder of this part provides an overview of these goals, including the scope, methodology, and contributions developed herein.

0.1 Distributed Computing Systems

Typically, a distributed system is a collection of multiple processes, running on different hosts, working to achieve a common goal. Then, distributed computing refers to the execution of distributed algorithms on distributed systems. A distributed algorithm executes as a collection of sequential processes, all executing their part of the algorithm without centralized control.

To coordinate their works, they use communications through some links. Such an algorithm receives, manipulates and outputs distributed data. A configuration of the system is a vector of the states of all processes together with messages in transit on all links.

In the sequential models, determinism is implied. It means that given a set of inputs, we can predict the execution of a deterministic algorithm. Nevertheless, determinism is not required in the execution of distributed algorithms. Since, processors may have different speeds and links may have different delays, the different runs of the same algorithm on the same set of inputs will have different executions. So, it is a hard task to predict the execution of a distributed algorithm. Still, many paradigms in distributed computing system including: *Election, naming, broadcast, mutual exclusion, synchronization and fault-tolerance*.

In distributed computing systems, the network is associated with some assumptions about the knowledge of its size, its topology, if the nodes are identified and so on. These information may be computed in a sequential manner and then diffused in order to be shared. That is, using pre-processing tasks for example. In addition, the computing system is characterized by some constraints about its communication system. In such systems, a problem is often formalized following some properties (specifications) as fairness, liveness, termination, closure and convergence. To prove the correctness of a given solution, we must prove that the corresponding algorithm satisfies all the corresponding properties under the system's assumptions. By contrast, to prove that some problem is unsolvable, it suffices to prove that for all its potential solutions, it is impossible to satisfy at least one of its properties. In this case, we *relax* some properties of the problem and/or assumptions about the distributed system.

The first classification of the distributed systems is based on the clock and the message transfer delays, we can find three principal classes:

- Synchronous system: the system is equipped with a total clock which controls the stages of calculations of each process. The speed of processes and the message transfer delays are bounded (known bounds).
- Asynchronous system: the system does not have a total clock. No knowledge is specified on the speed of processes nor on message transfer delays.
- Partially synchronous system: it is an asynchronous system which converges to a synchronous one.

0.2 Fault-tolerance

The growth of the distributed systems, mainly the use of a great number of computation units, increases the probability that some of these units break down during the execution of distributed algorithm leading to inconsistent executions. Algorithms which continue to function even after failure of some of the components must be implemented to guarantee the coherence of the system and to avoid the restarting of the treatments after each failure. These algorithms are qualified as fault-tolerant algorithms. Now, we give some terminology and techniques related to fault-tolerant computing.

0.2.1 Terminology

Here, we present the basic concepts related to processing failures [TS02].

Availability. Is the ability of a system to be used immediately. Such a property refers to the probability that the system is operating correctly at any given instant in time. Then, a system is said to be highly available if it is working almost usually.

Reliability. A system is said to be reliable if it can run continuously without failure. Compared to the availability property, the reliability property is related to some time interval instead of an instant in time. If a system breaks down for one millisecond every hour, it has an availability of over 99.9999 percent, but it still highly unreliable. For a system that never crashes but breaks down for two weeks every August, its reliability is high but its availability is only 96 percent.

Resilience. Such a factor deals with the degradation of the performances.

Dynamic Adaptability. To increase the availability of the system, it must be able to adapt itself to changes.

Safety. Deals with the ability of a system to operate correctly in the situation when it failed temporarily. For example, control systems such as those used to control nuclear power plants, are required to provide a high degree of safety.

Maintainability. This property refers to how easy a failed system can be repaired. That is when failures can be detected and repaired automatically. Such systems satisfy a high degree of availability.

Note, however, that the attribute of most significance for fault-tolerance is reliability and to some extent availability. In this thesis, we consider such attributes and in some way maintainability.

0.2.2 Fault-tolerance Techniques

A system is said to be failed when it cannot satisfy its tasks. For a distributed system devoted to provide some services, such a system fails if one or more of those services cannot be completely furnished. Since finding out the causes of a failure, then removing such failures is an important field in distributed computing systems, it is subject to much research in computer science. That is strongly related to what are called **dependable systems**. Building these kind of systems requires techniques to control faults. In [Lap92], a distinction between preventing, removing and forecasting faults is given. In this thesis, we will be interested in **fault-tolerance** issue. A system is said to be fault-tolerant if it can provide its services even in the presence of faults. Most works in the literature use the following techniques to improve the reliability of distributed computing systems.

Failure Models. In unsafe networks, we need to clarify some properties not used in a safe network sometimes referred as dynamic properties. Generally one can distinguish three types of failures in a distributed system:

- Final failure (crash,permanent): In this case, the process stops definitively functioning (physical breakdown).

- Byzantine failure(arbitrary): In this case, the process is carried out but moved away from the specifications (virus).

- Transient failure (temporary): In this case, the process stops its correct functioning due to a crash or byzantine failure and becomes correct again in a finite time: Physical crash followed by a repair, disconnection followed by a connection.

After determining the type of the fault that may occur, which is a difficult task, we need to adapt the specification of the problem to be solved in the presence of such faults. Byzantine faults are not considered in this thesis.

In an asynchronous system various problems remain extremely difficult or unresolved in the presence of crashes. One of the significant results, commonly called impossibility of *Fischer, Lynch and Paterson* [FLP85] about the *consensus* problem: *' There exists n processes with each one an unspecified initial value. These processes must agree on a common value '*. Initially, input values are held from some known set of values and output values are undefined. The specifications of this problem are as follows:

- Termination: any correct process must end up deciding,

- Validity: if a process decides a value *Val*, then *Val* is the initial value of the one of the processes,

- Agreement: two correct processes cannot decide on a different value.

The result of FLP, is expressed as follows: *'there is no any deterministic* [1] *algorithm which solves the consensus in an asynchronous message passing system with one crash failure of a process'*. This impossibility is explained mainly by the fact that in an asynchronous message passing system, it is impossible to distinguish between a crashed process and an only very slow one. This problem presents a great importance in the distributed systems since of much of other problems are reducible to this problem [MW87]. The basic approaches to solve this problem consist in introducing some weak forms of synchrony [FLM85, CT96].

Redundancy. The well-known strategy to design reliable systems, composed of multiple components, is the replication of all its components or just some of them. In fact, the key technique for masking faults is to use redundancy. Three types of redundancy are possible [Joh96, TS02]: Information redundancy, time redundancy and physical redundancy. For the first one, extra bits are added to recover from altered bits. For the second one, an action is performed, and then, if required, it is performed again. The use of transactions is an example of this technique. For the last one, extra material are added to make the system able to tolerate the break down or the malfunctioning of some components. This is the well-known technique. Intuitively, the increasing of the replication degree improves the reliability of the system but the cost of maintaining of such replications is expensive.

[1] Here, determinism is used instead of not probabilistic

Self-stabilization. Self-stabilizing [Dij74] algorithms ensure that starting from any configurations of the system, they converge to a legitimate state (configuration) in a finite time and remain in a legitimate state forever. Let faults that are bounded in time. Consequently, up to some time, which may be unknown but finite, the system may behave arbitrarily and end up in an arbitrary state. After that, the system behaves normally forever. In this case, we can say that self-stabilization protects against such failures, commonly named transient failures. In fact, we can consider the end up arbitrarily state as a started configuration for a given self-stabilizing algorithm. Thus, such an algorithm converges from a possibly erroneous state to a legitimate state. However, no further failures are allowed after the stabilization time. As stated, self-stabilization is fault-tolerance against faults that occur only at the beginning and their duration is limited.

Failure Detectors. Failure detectors have been introduced in the seminal paper of *Chandra* and *Toueg* [CT96] as an approved tool to solve the consensus problem [FLP85]. Informally, a failure detector is an oracle associated with each process that gives this process its mind, whether another process may have crashed or not. To solve the consensus problem the required failure detectors may make mistakes: They are named *unreliable failure detectors*. However, the use of failure detectors does not break the FLP impossibility. That is, their implementations require some synchrony from the system while the algorithm using their minds may be totally asynchronous. In other words, the use of failure detectors provides an abstraction of the needed synchrony. In addition, one of the main advantages of the use of failure detectors is that an application is not concerned how the failure detector is implemented.

Fault Containment. Fault containment has been cited as a technique for fault-tolerance [Nel90, Abb90]. Thereafter, it is introduced by [GG96, GGHP96, HH04] as an approach to ensure that the system is self-stabilizing and during recovery from a single transient fault, only a small number of processes will be allowed to execute recovery actions. The author presented examples to illustrate the possibility to design efficient fault-containment self-stabilizing protocols. Such algorithms contain the effects of limited transient faults while retaining the property of self-stabilization. Thus, fault containment aims to minimize the fault impact in order to reduce as much as possible their failure scope which can be any computational problem subject. As we shall see, the failure locality refines such a scope.

Local Fault-tolerance. The goal of local fault-tolerance is to isolate the scope of faults to minimal neighborhoods of impact. This is an alternative to avoid the use of redundancy which can be prohibitively expensive for some distributed systems and may be computationally impossible to achieve for others. Failure locality becomes a property of algorithms [CS92, CS96, KP00]. That is, an algorithm has a failure locality l, if the failure of any node v only affects nodes with distance at most l from v. Formally, this is equivalent to fix the radius of the set of processes possibly disrupted by a given fault to l and then such a radius can be viewed as a metric to quantify the degree of fault-tolerance [PS04].

0.3 Focus and Scopes of this Thesis

The focus of this thesis is to design distributed algorithms which run on an unreliable network and tolerate the processes failure using local knowledge. The motivation for this study is that the solution found in such an environment would be easily translated in other more environments with simple changes.

In this thesis we study the following aspects:

- **Models/Algorithms:** We start the study of a fault-tolerance in a distributed computing with the use of a local computations model and particularly graph relabeling systems. We give formalizations of a different faults classes in this model and we propose algorithms to solve some problems, including the spanning tree computations, resolving of conflicts. Since, the most known models to deal with failures are time-based models and communications play a great role in the design of protocols for these models, we introduce the failure detectors module to mask the abstraction of such notions in the local computations model. For the maintenance application, we propose both protocols in local computations and message passing models.

- **Experimentation:** We deal with three phases. First, we are interested to show interactively the execution of self-stabilizing distributed algorithms encoded by means of local computations. Second, Visidia is improved by an unreliable failure detector. So, some tests and measures are realized to determine the power of such a protocol and its possible use. Third, the improved platform allows us to design fault-tolerant distributed algorithms and to prototype distributed applications. Most of algorithms presented in this thesis are implemented.

- **Graph Structure:** Since the structure of the network especially its connectivity is a factor of its reliability, we propose a protocol to test the 2-vertex connectivity of a graph. Then, we extend such a protocol to compute a structure to deal with the maintenance of a forest of spanning trees in the presence of crash failures. The second use of a graph structure is to help us to design efficient algorithms. We begin by the use a cover of free-triangle graphs to improve the implementation of the self-stabilizing resolving of conflicts algorithm. Then, we exploit the connectivity knowledge to implement an efficient algorithm to maintain a spanning tree for k-connected graph in the presence of $k-1$ consecutive failures.

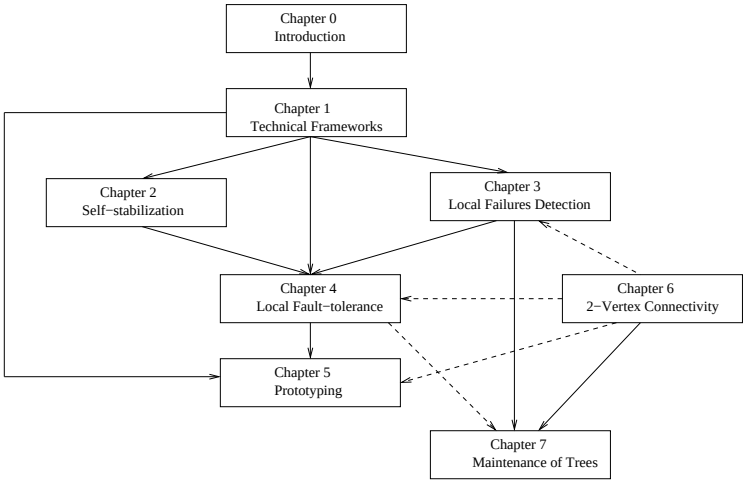


Figure 1: Chapters Dependencies

Chapter 1

Technical Frameworks and Tools

Models are used to denote some abstract representation of distributed computing systems. Specifically, we need models to represent the network, models to encode distributed algorithms and software platforms to test, to simulate and to validate the proposed solutions. Accordingly, comprehension, study and analysis of distributed algorithms require the seek of models which make as easy as possible to express them and to encode them with the following characteristics:

- Intuitive: to develop them and teach them,
- Formal: to prove their correctness using mathematical tools,
- Practical: to test and validate them by a simple implementation.

As a benefit, the study of some problems in high models allows us to deduce some properties on other less abstract models.

A distributed system (network) is often represented as a connected, undirected graph G denoted by (V, E): A node in V represents a process and an edge in E represents a bidirectional communication link. In addition, some assumptions are given on the structure of the graph and on the functioning of its components. In general, many studies deal with complete graphs, rings and trees, they consider anonymous or identified networks, they deal with the knowledge of each node about the size and/or the diameter of the network to name a few. Beyond this, these systems are based on the cooperation of multiple components, thus much attentions are gained on the mechanisms used to coordinate the work of such components. In particular, we consider communications that are fully and clearly expressed or not. The relevant literature consists of works that use two types of models:

- **Models using abstract communications**: Graph relabeling systems [LMS99] and state model [Dij74].
- **Models using explicit communications**: Message passing model [AW98, Tel00] and shared memory [Lyn96].

In this thesis, we will use: The graph relabeling systems and the message passing model. In both models, a distributed algorithm for a collection $V = \{v_1, \cdots, v_N\}$ of processes is a collection of local algorithms (or programs), one for each process in V. To avoid confusion

between the states of a single process and the states of the entire algorithm, commonly named the global state, the latter will from now on be called *configurations*.

These sections present elements of the formal frameworks and some tools we will use to construct fault-tolerant distributed algorithms. In Section 1.1, we give definitions about the graph theory. Section 1.2 contains the basic definitions and terminology used to describe graph relabeling systems as the first model used to encode distributed algorithms presented in this thesis. In section 1.3, we describe the second model which is the asynchronous message passing model. We show in Section 1.4 what means comparing complexities of algorithms in different models. In section 1.5, we give the place of the used models regardless of the timing in distributed computing systems. Section 1.6 presents briefly Visidia, a software tool used to validate some results of the thesis.

Reader already familiar with these concepts and definitions may safely proceed to subsequent chapters and consult the definitions and notations herein only as needed.

1.1 Graphs

We introduce some terminologies and definitions about graphs. A graph is a collection of points called the nodes of the graph where some of them are connected by lines (adjacent) called the edges.

In each class of graphs we can use the following structure (V, E) to design a graph G, where V is the set of nodes and $E \subseteq \{\{u, v\}$ such that $u, v \in V$ and $u \neq v\}$ is the set of edges. For a graph $G = (V, E)$, we use sometimes the notations V_G, E_G to denote respectively the set of nodes of G and the set of edges of G.

In the sequel we will use graphs to model computational systems. The nodes of a graph represent the computers (or processors) and the edges correspond to the communication links between pairs of processors. Since our formalizations are based on undirected graphs, we use (u, v) instead of $\{u, v\}$ to denote an edge. So, the notations (u, v) and (v, u) are equivalent. Let $(u, v) \in E_G$, we say that u is a neighbor of v.

For any set A, we write 2^A to denote the set of all finite subsets of A. We use $\#M$ to denote the cardinality (or the size) of the set M.

Definition 1.1.1 *A path p in G is a sequence $(v_0, ..., v_l)$ of nodes such that for each $i < l$, $(v_i, v_{i+1}) \in E$. The integer l is called the length of p. Then, a path $p = (v_0, v_1, ..., v_k)$ of non-zero length through the graph such that $v_0 = v_k$, is called a cycle. A path $P = (v_0, ..., v_l)$ is simple if it does not contain cycles.*

Definition 1.1.2 *The set of neighbors of a node u is denoted by $NG(u) = \{v \in V/(u, v) \in E\}$. The degree of a node u, denoted* **deg(u)**, *is the number of neighbors of u. Then, $deg(u) = \#\{\{u, v\} \in E$ such that $v \in V\}$. The degree of G is $deg(G) = max\{deg(v)$ such that $v \in V\}$.*

Definition 1.1.3 *The l-neighborhood function $NG_l(u)$ is the set of nodes reachable within at most l hops of node u. For $l > 0$, this function can be defined recursively as follows:*

- $NG_1(u) = NG(u)$,
- $NG_{l+1}(u) = NG_l(u) \cup \{\forall v \in V : \exists w \in NG_l(u)$ *and* $(v, w) \in E\}$.

Definition 1.1.4 *A ball centered on u with radius l is the set $B_l(u) = \{u\} \cup \{v_j \in V /$ there exists a path $(v_0, ..., v_j)$ in G with $v_0 = u$ and $j \leq l\}$. Formally, this is equivalent to say that $v \in V$ is a $l-$neighbor of u if v is in $B_l(u)$. For simplicity, we will write $B(u)$ to mean the ball centered on u with radius* 1.

Definition 1.1.5 *We use $dist(u, v)$ to denote the length of the shortest path in G between u and v. Then, the diameter of a graph G, denoted by* **diam(G)** *is the longest distance among all pairs of nodes in G. Formally, $diam(G) = max\{dist(u, v)$ such that $u, v \in V_G\}$.*

Definition 1.1.6 *We say that the graph $G' = (V', E')$ is a sub-graph of the graph $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$.*

Let $V' \subseteq V$, we denote by $G|V'$ the restriction of G to the sub-graph $(V', \{(u, v) \in E \cap V'^2\})$. We also denote by $G \setminus V'$ the graph $G|(V \setminus V')$ [1]. When V' is a singleton $V = \{v\}$, by abuse, we will use the notations $G|v$ and $G \setminus v$.

Definition 1.1.7 *A graph $G = (V, E)$ is a complete graph iff $E = V^2$. In this case, we say that G is a fully connected graph. Otherwise, it is sparse.*

Definition 1.1.8 *A graph is connected if for any pair of its nodes, there exists a path between them.*

Definition 1.1.9 *We denote by K_G the connectivity of a graph $G = (V, E)$ which is the minimum size of any set $V' = \{v_1, ..., v_k\} \subseteq V$ such that $G \setminus V'$ is not connected.*

Theorem 1.1.10 (Menger) *a graph $G = (V, E)$ is a k-connected graph if for any pair $u, v \in V$ there exists k disjoint paths linking u and v.*

Definitions 1.1.11

- *A tree $T = (V_T, E_T)$ is a connected graph that have no cycle and a chosen node v_0 called the root.*
- *As usual, we associate with a tree, a partial order $\leq$ on V_T defined by $u \leq v$ iff there exists a simple path $v_0, v_1 \cdots, v_l$ in T such that $u = v_i$ and $v = v_j$ with $i \leq j$. If $j = i + 1$, u is the "Father" of v and v is a son of u.*
- *We denote by $Sons(u)$ the set of the sons of u. In the sequel, we assume that the list "Sons" of u is ordered.*
- *A node $u \in V_T$ with no son is called a leaf of T.*
- *Then, a spanning tree $T = (V_T, E_T)$ of a graph G is a tree such that $V_T = V_G$ and $E_T \subseteq E_G$. The tree can be defined also by $T(Father, Sons)$.*

Definition 1.1.12 *A graph morphism φ between two graphs $G = (V, E)$ and $G' = (V', E')$ is a surjective mapping $\varphi : V \to V'$ such that $\forall\ (u, v) \in E$: $(\varphi(u), \varphi(v)) \in E'$ and $\forall\ (u', v') \in E'$, $\exists (u, v) \in E$ such that $u' = \varphi(u)$ and $v' = \varphi(v)$.*

[1] "\" denotes the difference between two sets

1.2 Graph Relabeling Systems (GRS)

A labeled graph is a natural model to study computer networks. Nodes of graph correspond to computers or processors, its edges stand for communication links, and its labeling represents the network state. In [LMS99], the authors proposed a powerful model to encode distributed algorithms. This model is based on the use of graph relabeling systems. It offers general tools to encode distributed algorithms, to prove their correctness and to understand their powers. In this thesis, we will use some restricted version of this model. At any time we specify the kind of the rewritings that we will use.

1.2.1 The Model

Here we present some fundamental basis of the graph relabeling systems and how we can use them to encode distributed algorithms.

Definitions 1.2.1 *Let Σ be an alphabet.*

- *A* labeled graph *is a couple (G, L) where G is a graph and L is a* labeling function *i.e. a mapping from $V_G \cup E_G$ to Σ.*
- *We denote by $\mathcal{G}_L$ the set of all labeled graphs (G, L).*
- *The labeled graph (H, L') is a sub-labeled graph of (G, L) , if H is a sub-graph of G and L' is the restriction of L to $V_H \cup E_H$.*

In the sequel, unless stated otherwise, we assume that all the used labeling functions share the same alphabet.

Definitions 1.2.2

- *A labeled graph morphism φ: $(G, L) \longrightarrow (G', L')$ is a graph morphism from G to G' which preserves the labeling, that is, for any $x \in V_G$, $L'(\varphi(x)) = L(x)$.*
- *A labeled sub-graph (G', L') of (G, L) is an* occurrence *of (H, L_H) in (G, L) if there exists an isomorphism [2] φ between H and G' such that $\forall x \in V_H$, $L_H(x) = L'(\varphi(x))$.*

Definition 1.2.3 *A graph relabeling rule is a triple $R = (H, L', L_H)$ such that (H, L') and (H, L_H) are two labeled graphs. The labeled graph (H, L') is the* precondition part *of R and the labeled graph (H, L_H) is its* relabeling part.

Definitions 1.2.4 *Given a labeled graph (G, L_i),*

- *An algorithm $\mathcal{R}$ is a finite set of relabeling rules.*
- *An $\mathcal{R}$-*computing step *is a couple of label functions (L_i, L_{i+1}) such that there exists a sub-labeled graph (G', L') of (G, L_i) and a rule $R = (H, L', L_H) \in \mathcal{R}$ satisfying : (G', L') is an occurrence of (H, L_H) in (G, L_i), $\forall x \in V_G \setminus V_H$, $L_{i+1}(x) = L_i(x)$ and $\forall x \in V_H$, $L_{i+1}(x) = L_H(x)$. In other words, the labeling L_{i+1} is obtained from L_i by modifying all the labels of the elements of G' according to the labeling L_H. Such a computing step will be denoted by $L_i \underset{R}{\longrightarrow} L_{i+1}$.*

[2] An isomorphism is a bijective morphism.

We say that the *computations in* G *are local* if a node $u \in V_G$ can only act on the set of its $l-$neighbors in a ball of a fixed radius l. Formally, this is equivalent to fix the diameter of the graph H of the relabeling rules $R \in \mathcal{R}$ to $2l$.

Any computation in G is a relabeling sequence $(L_0, L_1, \cdots, L_n)$ such that $\forall\, 0 \leq i < n,\ (L_i, L_{i+1})$ is a computing step. We use the notation $L \xrightarrow[\mathcal{R},p]{} L'$ to say that there exists a local computation of length p starting from the label L to the label L' i.e. there exists a computation $(L_0, L_1, \cdots L_p)$ such that $L_0 = L,\ L_p = L'$. If the number p is not known, we will use $L \xrightarrow[\mathcal{R},*]{} L'$. The labeled graphs (G, L_i) denote the *configuration* or the global state of the system at this stage of computation. Usually, (G, L_0) is called *initial configuration*.

The computation stops when the graph is labeled in such a way that no relabeling rule can be applied. A labeled graph (G, L) is said to be $\mathcal{R}-$*irreducible* (or irreducible when it is clear from the context) if there exists no occurrence of (G_R, L_R) in (G, L) for every relabeling rule R in $\mathcal{R}$.

Definition 1.2.5 *A computation in* G *is terminated if no relabeling rule can be applied on the current labeling. In this case, the corresponding labeled graph* (G, L') *is a result of the algorithm. We denote by* $RES_{\mathcal{R}}(G)$ *the set of all the results of the algorithm* $\mathcal{R}(G)$.

For a sake of clarity, we will denote a graph relabeling system $\mathcal{R}$ using the following notation: $\mathcal{R} = (\Sigma, \mathcal{I}, \mathcal{P})$, where Σ is an alphabet for the used labels, $\mathcal{I}$ is s subset of Σ to denote the required initialization labeling and $\mathcal{P}$ is the set of the $\mathcal{R}$-relabeling rules.

A graph relabeling system *with priorities* is a 4-tuple $\mathcal{R} = (\Sigma, \mathcal{I}, \mathcal{P}, \succ)$ such that $(\Sigma, \mathcal{I}, \mathcal{P})$ is a graph relabeling system and $>$ is a partial order defined on the set P called the *priority relation*. A $\mathcal{R}$-relabeling step is then defined as a 5-tuple (G, L, R, φ, L') such that R is a relabeling rule in $\mathcal{P}$ and φ is both an occurrence of (G_R, L_R) in (G, L) and an occurrence of (G_R, L'_R) in (G, L') and there exists no occurrence φ' of a relabeling rule R' in $\mathcal{P}$ with $R' \succ R$ such that $\varphi(G_R)$ and $\varphi(G_{R'})$ intersect in G.

Let (G, L) be a labeled graph. A context of (G, L) is a triple (H, μ, ψ) such that $H(N, L, \mu)$ is a labeling graph and ψ an occurrence of (G, L) in $H(N, L, \mu)$. A relabeling rule with *forbidden contexts* is a 4-tuple $R = (G_R, L_R, L'_R, F_R)$ such that (G_R, L_R, L'_R) is a relabeling rule and F_R is a finite set of contexts of (G_R, L_R).

A graph relabeling system with forbidden contexts is a triple $\mathcal{R} = (\Sigma, \mathcal{I}, \mathcal{P})$ defined as a graph relabeling system except that the set $\mathcal{P}$ is a set of relabeling rules with forbidden contexts. A $\mathcal{R}$-relabeling step is a 5-tuple (G, L, R, φ, L') such that R is a relabeling rule with forbidden contexts in $\mathcal{P}$ and φ is both an occurrence of (G_R, L_R) in (G, L) and an occurrence of (G_R, L'_R) in (G, L'), and for every context (H_i, μ_i, ψ_i) of (G_R, L_R), there is no occurrence φ_i of $H_i(N_i, L_i, \mu_i)$ in (G, L) such that $\varphi_i(\psi_i(G_R)) = \varphi(G_R)$.

For more formal definitions and examples on relabeling systems, the reader is referred to [LMS99]: They proved that for each relabeling system with forbidden context we can associate an equivalent relabeling system with priority. Note, however, that a forbidden context is usually used to encode a control structure part of an algorithm.

1.2.2 The Program

In this thesis, we consider only relabeling between neighbors. That is, each one of them may change its label according to rules depending only on its own labels, the labels of its neighbors and the labels of its corresponding edges. We assume that each node distinguishes its neighbors and knows their labels.

The program is encoded with graph relabeling system $\mathcal{R} = (\Sigma, \mathcal{I}, \mathcal{P})$. The labels of each process represent the value of its variables. Each rule in the set $\mathcal{P}$ is an action which has the following form:

$$\text{R1} : \textbf{RuleN}\{Precondition\}\{Relabeling\}$$

The label R1 is the number of the rule and the label **RuleN** is the name of the rule. The component $Precondition$ of a rule in the program of v_0 is a Boolean expression involving the local labels of v_0. The $Relabeling$ component of a rule of v_0 updates one or more labels its local labels. A rule can be executed only if its precondition evaluates $true$. The rules are atomically executed, meaning that the evaluation of a precondition and the execution of a corresponding relabeling, if the precondition is $true$, are done in one atomic step.

Often the notion of relabeling sequences corresponds to a *sequential* execution. For a local computations model we can define equivalent parallel rewritings, when two consecutive relabeling steps concerning non-overlapping balls may be applied in any order. They also can be applied concurrently.

1.2.3 Proof Techniques

Graph relabeling systems provide a formal model to express distributed algorithms. The aim of this section is to show that this model is suitable to study and to prove properties of distributed algorithms.

Termination. A graph relabeling system $\mathcal{R} = (\Sigma, \mathcal{I}, \mathcal{P})$ is *noetherian* if there is no infinite $\mathcal{R}$-relabeling sequence starting from a graph with initial labels in $\mathcal{I}$. Thus, if a distributed algorithm is encoded by a noetherian graph relabeling system then this algorithm always terminates. In order to prove that a given system is noetherian we generally use the following technique.

Let $(S, <)$ be a partially ordered set with no infinite decreasing chain (that is every decreasing chain $x_1 > x_2 > \cdots > x_n > \ldots$ in S is finite). Then, we say that $<$ is a *noetherian order*. It is compatible with $\mathcal{R}$ if there exists a mapping f from $\mathcal{G}_L$ to S such that for every $\mathcal{R}$-relabeling step $(G, L) \to (G, L')$ we have $f(G, L) > f(G, L')$.

It is not difficult to see that if such an order exists then the system $\mathcal{R}$ is noetherian: since there is no infinite decreasing chain in S, there cannot exist any infinite $\mathcal{R}$-relabeling sequence. For our examples, the set S will be a set $\mathbb{N}^p$ where p is an integer and the ordering relation is defined by $(x_1, \ldots, x_p) >_p (y_1, \ldots, y_p)$ which means that there exists an integer j such that $x_1 = y_1, \ldots, x_{j-1} = y_{j-1}$, and $x_j > y_j$.

Correctness. In order to prove the correctness of a graph relabeling system, that is the correctness of an algorithm encoded by such a system, it is useful to exhibit (i) some *invariant properties* associated with the system (by invariant property, we mean here some property of

the graph labeling that is satisfied by the initial labeling and that is preserved by the application of every relabeling rule) and (ii) some properties of irreducible graphs. These properties generally allow to derive the correctness of the system.

1.2.4 Complexity Measures

The execution time of a computation is the length of the corresponding relabeling sequence. Thus, a measure of the time complexity of a distributed algorithm, or of a graph relabeling system, will be the execution time of the longest computation of the algorithm. A measure of a space complexity of a distributed algorithm is the size of the label of a node of the graph. Obviously, the used concrete memory is the size of all the labels in the graph.

1.2.5 Implementation

Let $G = (V, E)$ be a graph, v a vertex in V and l some positive integer. We say that v is *locally elected* in $B_l(v)$ if there is a way to distinguish v from all the other vertices of $B_l(v)$. This property is necessary if we need to ensure that no two vertices in a distance less or equal to l change their labels simultaneously. In this work we will use randomized procedures from [YM02] to obtain this property in balls of radius 1.

Randomized local elections are used to ensure that no two adjacent processes change simultaneously their states. In fact, the model of distributed computation is an asynchronous distributed network of processes which communicate by exchanging messages. To overcome the problem of certain non-deterministic distributed algorithms as well as to have efficient and easy implementations, we use randomization [BGS94, Tel00, HMRAR98]. General considerations about randomized distributed algorithms may be found in [Tel00] and some techniques used in the design and for the analysis of randomized algorithms are presented in [MR95, HMRAR98, BGS94].

In [MSZ00, YM02], the authors have investigated randomized algorithms to implement distributed algorithms specified by local computations and graph relabeling systems. Intuitively, each process tries at random to synchronize with one of its neighbors or with all of its neighbors depending on the model we choose, then once synchronized, local computations can be done.

As we shall see, we use these randomized procedures to analyze and to implement an algorithm for resolving conflicts in the context of self-stabilizing systems. We use them also to implement distributed algorithms using the *Visidia* software.

There are three types of local computations. The implementation of these local computation in an asynchronous message passing system is based on the use of randomized synchronizations. A randomized synchronization procedure is associated with each type of local computation given in the following:

- *Rv (Rendezvous)* :
  - Synchronization : There is a *rendezvous* between vertex v and its neighbor $c(v)$, we say that v and $c(v)$ are synchronized.
  - Computation : Then v and $c(v)$ exchange messages and the labels attached to nodes of H_2 (the complete graph with 2 nodes) are modified according to some

rules depending on the labels appearing on K_2.

- *Lc1 (local computation 1)* :
  - Synchronization : The vertex v is elected in the star centered on v denoted b_v.
  - Computation : The computation is allowed on b_v : the center v collects the labels of its neighbors and changes its label. The label attached to the center of a ball is modified according to some rules depending on the labels of the ball, labels of the leaves are not modified.
- *Lc2 (local computation 2)* :
  - Synchronization : The vertex v is elected in the star centered on v of radius 2 denoted b_v.
  - Computation: The computation is allowed on b_v : The nodes of b_v exchange their labels, and the center updates its state and the states of its neighbors. The labels attached to the center and to the leaves of a ball may be modified according to some rules depending on the labels of the ball.

When a distributed algorithm is implemented using such procedures, its time complexity may be measured in terms of the needed synchronizations (or local elections).

1.2.6 Example of a Distributed Spanning tree Computation

Let us illustrate the use of graph relabeling systems for the computation of a spanning tree in a network with a pre-chosen *root*. Consider the following graph relabeling system used to compute a distributed spanning tree of a given graph $G = (V, E)$. Every node $u \in V$ has for label the triplet:

- $Span(u)$: the state of u that can have only 2 values: A to mean that v has been included in the tree or N when it is not yet in the tree,
- $Father(u)$: is the father of u in the spanning-tree.
- $Sons(u)$: the ordered list of its sons in the tree.

For the label $Father$, it may be seen as a port number. In this case, each node u is equipped with ports numbered from 1 to $deg(u)$. Further, we write $Father(u) = \perp$ to mean that v has no defined father. When starting the algorithm, we choose a node v_0 as the *root* of the tree. The initial label function L_0 is defined by :

- $Span(v_0) = A$, $Father(v_0) = \perp$, $Sons(v_0) = \emptyset$,
- $\forall u \neq v_0$, $Span(u) = N$, $Sons(u) = \emptyset$, $Father(u) = \perp$,

The algorithm is described by a relabeling system $\mathcal{R}$ with one rule R1 defined on $\Sigma = \{\{N, A\} \times \{1 \cdots deg(G) \cup \{\perp\}\} \times 2^{deg(G)}\}$:

At any step of the computation, when a N-labeled node u finds a neighbor v with $Span(v) = A$, this node u may decide to include itself in the tree by changing $Span(u)$ to A. Moreover, $Father(u)$ is set to v and the node v adds u in its sons list. So at each

Algorithm 1 Distributed spanning tree algorithm in the local computations model

R1 : **Spanning rule acting on 2 nodes** $v,\ u$

Precondition :

- $Span(u) = N$
- $\exists\, v \in B(u), Span(v) = A$

Relabeling :

- $Span(u) := A$
- $Father(u) := v$
- u is added to $Sons(v)$

computation step, the number of N-labeled nodes decreases by 1. The computation finishes when all the nodes v are such that $Span(v) = A$. And obviously we have a spanning tree of G rooted at v_0 defined by the second components (or the third components) of the labels of the nodes.

To prove that the result is a spanning tree of a graph G, it suffices to prove the following invariants:

Lemma 1.2.6 *Let (G, L) be an initial labeled graph. Let (G', L') be the graph obtained after the application of a finite number of relabeling steps of system $\mathcal{R}$. Then, (G', L') satisfies the following properties:*

(I1) all N-labeled nodes are $\perp$-parent.

(I2) each parent is an A-labeled node.

(I3) the sub-graph induced by the $node\text{-}father$ edges is a tree.

(I4) each graph in the set $RES_{\mathcal{R}}(G)$ is a spanning tree.

Proof.

(I1) Let us prove this property by induction on the size of the relabeling sequence. Initially the property is true. Now assume that the property is true for a graph (G, L) obtained from the initial graph after k applications of the rule $R1$, we will show that the property remains true at step $k + 1$. In the k^{th} step, all the N-labeled nodes are $\perp$-parent. If we apply $R1$ on one of such nodes, we can apply it to change both the labels $Span$ and $Father$. That is, when a node changes its $Father$ from $\perp$ to another value, it changes in the same time its label $Span$ from N into A. Hence, the property is still true.

(I2) A the beginning, the property is true since there is only one vertex labeled A without parent(the root). Suppose the property is true after k relabeling steps. We will prove that the property remains true at the $(k+1)^{th}$ step. In the k^{th} step, all vertices labeled A have their parent labeled A. We cannot apply any rules between nodes labeled N. If we apply $R1$ on the vertex labeled N, it will be labeled A changing its parent: It will be connected to a A-labeled vertex as its parent. Thus the property remains true.

(I3) By (I2), the A-labeled vertex, except the root, is connected exactly with one vertex labeled A by a marked edge. So the sub-graph induced by the A-labeled nodes, the root which is labeled A and the marked edges has no cycle.

(I4) By (I1), there exists only one vertex labeled A with undefined parent. By (I2), all its neighbors labeled A and connected to it by a marked edge are labeled also A. We can apply $R1$ on the other N-labeled vertices to obtain vertices which check (I2) and then (I3). So, if G' is an irreducible graph obtained from a graph G, then all its nodes are labeled A, it contains exactly one node with undefined father and all others have exactly one parent.

□

To prove that the algorithm finishes, it suffices to see that at each computation step, the number of N-labeled nodes decreases by 1 to reach 0. That is, to prove the termination of a graph relabeling system and then the termination of the distributed algorithm it encodes it suffices to prove that it is noetherien [LMS99].

The following property follows from the previous results:

Property 1.2.7 *Given a graph $G = (V, E)$ and a chosen node v_0. A spanning tree of G rooted at v_0 can be computed by applying $\#V - 1$ rules.*

1.3 Message Passing Model (MPS)

For the designers of distributed applications, the message passing model is the more natural model to encode distributed algorithms. This section presents the formal model for message passing model [AW98, Tel00]. Then, we show the pseudo-code conventions for describing message passing algorithms and finally we define the basic complexity measures.

1.3.1 The Model

The system is modeled by a connected simple graph where each node represents an autonomous entity of computation (e.g. *thread*, process, machine) and each edge a communication channel. An edge is present between two nodes if their corresponding processors may communicate by sending messages over this edge.

Since the processor is identified with a particular node in the graph, edges incident to u are labeled using arbitrary set of numbers in the set $1 \cdots r$ where r is the degree of u. That is,

Definition 1.3.1 *Each node u is equipped with ports, numbered from 1 to $deg(u)$, which will be used to distinguish between and to communicate with neighbors. The attribution of these numbers is completely arbitrary and does not depend on the identities of the neighboring nodes.*

The local program at each node consists of a set of variables (state) and a finite set of local actions that a node may make. They include modification of local variables, send messages to and receive messages from each of its neighbors in the corresponding graph topology.

More formally, each node u is modeled using a set of states. Node u's action takes as input a value of some states of u, messages sent by the neighbors. Then, it produces as output a new value of some states and also at most one message for each link between $1 \cdots deg(u)$. Messages previously sent by u and not yet delivered cannot influence u current step.

Definition 1.3.2 *At any time, a configuration or a global state of the system is a vector C composed of the states of all the processors together with m sets- one set of every link- of messages in transit on*

that link at this time. An initial configuration is a vector of the initial states of all the processors when all the links are free.

Definition 1.3.3 *The system is said to be asynchronous if processes operate at arbitrary rates and message transfer delays are unbounded, unpredictable but finite.*

Definition 1.3.4 *A computation $\sigma = \sigma_1\sigma_2....$ of an algorithm $\mathcal{R}$ is a finite or infinite sequence of configurations $C_1, C_2, ...$ with: For $i = 1, 2, ...$ the configuration C_{i+1} is reached from C_i by the computation step σ_i.*

Note, however, that during a computation step, one or more processors execute one action. When σ is finite and I denotes an initial configuration, the resulting configuration $\sigma(I)$ is said to be *reachable* from I and is said to be *accessible*.

To model a computation in a message passing system, we introduce two kinds of events.

Definition 1.3.5 *The computation event, representing a computation step of a processor u in which it applies action to its current accessible states.*

Definition 1.3.6 *The delivery event responsible of the deliverance of message m from processor u to processor v.*

Definition 1.3.7 *An execution segment α of a asynchronous message passing system is a finite or infinite sequence of the following form:*

$$C_0, \phi_1, C_1, \phi_2, C_2, \phi_3, \cdots$$

where C_i is a configuration and ϕ_j is an event for all $i \geq 0$ and $j \geq 1$.

So an execution shows the behavior of a system, which is a set of sequence of configurations and events. Such sequences depend of the task assigned to the system being modeled. Thus, they must satisfy some properties according to the tasks.

To define the termination of the distributed algorithm, we assume that each node's states includes a subset of states to denote *terminated* states. After the reach of such states, actions of the program maps terminated states only to terminated states. Thus,

Definition 1.3.8 *We say that the algorithm has terminated when all nodes are in the terminated states and all messages are delivered.*

In this thesis, we assume that, for a couple of neighboring vertices, the order of sending messages is the same as that of receiving them (FIFO).

1.3.2 Program

We will specify an algorithm in the message passing model as a "pseudo-code". That is, for each process algorithm will be described in an interrupt-driven fashion. The effect of each message is described individually, and each process handles the pending messages one by one in some arbitrary order. Some processes may also take some action even if no message is received. Actions that are listed are only actions that cause sending messages and/or states changes. The computation associated with each process done within a computation will be

described in style closed to typical pseudo-code for sequential algorithms. Some reserved words may be used to indicate some specific states.

In the pseudo-code, the local state variables of processor v_i will not necessary sub-scripted with i; but used in some proofs and discussion to avoid ambiguity. Comments will begin with $/*$ and end with $*/$.

In the following we give an example of a spanning tree computation algorithm described in pseudo-code. For a sake of clarity, in the explanation of the algorithms encoded in the message passing model we refer to a node instead of its corresponding port number.

1.3.3 Complexity Measures

Complexity measures in the message passing model are related to the number of messages and the amount of time. As the local computations model, we will focus on the worst-case. Such measures depend on the notion of the algorithm terminating or when algorithm reaches some specific configuration.

The message complexity of an algorithm encoded in the message passing model is the maximum of the total number of message sent during the all possible executions of this algorithm. For the time complexity, we approve the common approach assuming that the maximum message delay in any execution is one unit of time. Note, however, that such a measure is not used to prove the correctness of such algorithms. Then, to calculate the time complexity of an algorithm it suffices to take the running time until the termination.

A measure of a space complexity of a distributed algorithm is the size of the variables at a node of the graph. Obviously, the concrete memory used is the size of all the variables in the graph.

1.3.4 Example of a Distributed Spanning Tree Computation

Consider the following algorithm used to compute a distributed spanning tree of a given graph $G = (V, E)$. When starting the algorithm, we choose a node v_0 as *the root* of the tree.

Here, $Father(u) = \perp$ means that v has no defined father. At any step of the computation, when a node u, not yet in the tree ($in\text{-}tree(u) = false$) receives a <**tok** > message from its neighbor v, node u includes itself in the tree by changing its $in\text{-}tree$ to $true$. Moreover, in the same time, $Father(u)$ is set to v and u informs v to add it in its set of sons sending <**son**> message. At the reception of such a message, v adds u in its set of sons. So at each execution of the rule $T(1)$, the number of nodes not yet in the tree decreases by 1. The computation finishes when all the nodes u are such that $in\text{-}tree(u) = true$ and all the messages are treated. And obviously we have a spanning tree of G rooted at v_0 defined by the components $Father$ (or the components $Sons$) of the variables of the nodes. The root of the spanning tree is then the unique node with its father equal to $\perp$.

Property 1.3.9 *Given a graph $G = (V, E)$ and a chosen node v_0. A spanning tree of G rooted at v_0 can be computed by exchanging $(2\#E - (\#V - 1))$ <**tok**> messages and $(\#V - 1)$ <**son**> messages.*

Algorithm 2 Distributed spanning tree algorithm in the message passing model.

var $Father$: integer **init** $\perp$;
$in\text{-}tree$: Boolean **init** $false$;
$Sons$: set of integer **init** $emptyset$;
i, q : integer;

I : {For the root v_0 only, execute once:}
$in\text{-}tree(v_0) := true$;
for ($i := 1$ **to** $deg(r)$) **do** send<**tok**> via port i

T : {A message <**tok**> has arrived at u from port q}
1: **if** ($not\ in\text{-}tree(u)$)
$Father(u) := q$;
$in\text{-}tree(u) := true$;
send<**son**> via port q;
for ($i := 1$ **to** $deg(u)$) **do** send<**tok**> via port i

S : {A message <**son**> has arrived at u from port q}
$Sons(u) := Sons(u) \cup \{q\}$

1.4 Comparing Complexities and Models

It's important to take caution when comparing complexities of algorithms in different models. So, it's often not significant to compare time complexity of an algorithm encoded in the local computations model with an algorithm to solve the same problem and encoded in the message passing model for example. The problems are structured differently on these two models. But, it's meaningful to start to encode an algorithm in high model, to understand its power and to deduce some properties to help us to design it on other models. On another hand, it is practical to show that the efficiency in these models is not violated.

1.5 Timing in Distributed Computing Systems

There are two basic models of timing in distributed computing system : *Synchronous and Asynchronous*. In the synchronous model, the existence of a global clock is assumed, each process executes simultaneously one step of its program in each time step. In the asynchronous model, processes execute their programs at different speeds. In this model, the difference between the speeds of the processes is simulated with the use of a *scheduler (daemon)*. This mechanism allows us to explicit some synchronization constraints. At each step the scheduler activates one, all or a subset of enabled processes (their states are such that they can perform a step of their program). Thereby, the GRS model is asynchronous since several relabeling steps may be applied at the same time but we do not require that all of them have to be performed. For the MPS model, a system is said to be asynchronous if there is no fixed upper bound on the duration of both computation and delivery events.

In spite of many distributed applications use usual upper bounds on message delays and processor step times, it is often suitable to design algorithms that are independent of any particular timing parameters, namely asynchronous algorithms.

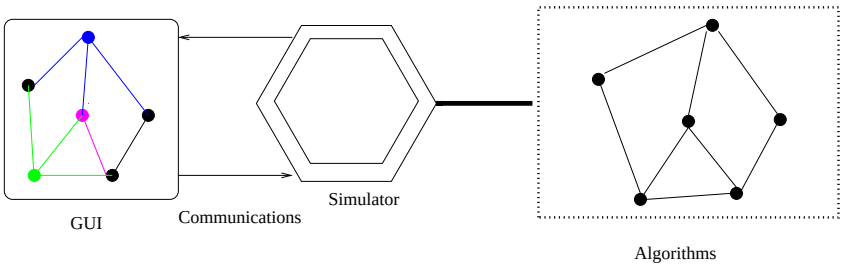


Figure 2: Visidia architecture

In the asynchronous message passing model, an execution is said to be *admissible* if each processor may take an infinite number of computations events, and every message sent is eventually delivered. Such requirements model the fact that processors do not fail and communication links are also reliable. It does not imply that the processors programs must contain an infinite loop. In this point of view, the termination of an algorithm is reached when actions don't change the processor states after a certain point.

1.6 *Visidia*, A Tool to Implement, to Visualize and to Simulate Distributed Algorithms

Visidia [MS, BMMS01] is a tool to implement, to simulate, to test and to visualize distributed algorithms. It is motivated by the important theoretical results on the use of graph relabeling systems to encode distributed algorithms and to prove their correctness [LMS99, Sel04, Oss05, Der06, Cha06]. Visidia provides a library together with an easy interface to implement distributed algorithms described by means of local computations. This platform is written in *Java* language. The distributed system of Visidia is based on asynchronous message passing system. In this tool the processes are simulated by *Java threads*. As such, a stage of computation is carried out after some synchronization achieved using probabilistic procedures [MSZ00]. As shown in Figure 2, Visidia is composed of three parts:

- The graphical interface part (*GUI*) to visualize the execution of the distributed algorithms,
- The *Algorithm* part where the distributed algorithms are executed,
- The *Simulator* part which ensures the visualization of the state of the distributed algorithm execution on the *GUI*.

Specifically, Visidia allows to construct a network as a graph where nodes represent processes and edges represent communication channels. Data and status of processes are encoded using labels which are displayed on the screen.

There are two implementations of Visidia. The first one is implemented in a single machine. In this version, the threads representing the processes of the computation are created on the same machine. As a thread is an independent execution flow, the execution of the algorithm is actually distributed. Each thread communicates with the simulator which transfers the events to the visualizer. The second one is an implementation on a network of machines. In this version, the threads are executed on a network of machines. One or several threads are

assigned to each physical machine participating in the computation. The visualization is done on the local host of the user. In such a model, there is no simulator. In fact, a node can send a message to a neighbor by itself.

To implement a relabeling system, Visidia provides high level primitives including the synchronization procedures and communication actions. Especially, three functions: $rendezVous()$, $starSynchro1()$, $starSynchro2()$ implementing the previous synchronizations. Moreover, two actions to express the communication between two neighborhood processes u, v : If u wants to send a message "msg" to v, it executes the action $sendTo(v, msg)$. On the other hand, the process v executes the action $receiveFrom(u)$ to get the message "msg".

We sketch in the following an implementation of the relabeling system given in Section 1.2.6, to encode a distributed computation of a spanning tree, using Visidia software. The presented program is based on the *Java* code used in the Visidia platform.

Algorithm 3 Encoding of algorithm 1 in the Visidia Software

```
while (run){
    // start of a synchronization
     neighbor = rendezVous();
    // exchange of states
    sendTo(neighbor,mySpan);
    neighborSpan = receiveFrom(neighbor);
    //execution of one rule
    if ((mySpan ==’N’) && (neighborSpan ==’A’)){
      mySpan =’A’;
      myFather = neighbor;
    }
    else
      if ((mySpan ==’A’) && (neighborSpan ==’N’)){
         mySons[neighbor] = true;
      }
    // end of a synchronization
    breakSynchro();
}
```

As illustrated in this program, each process tries to reach a synchronization with one of its neighbors. Note, however, that a node can try to reach synchronization with all its neighbors according to the type of the local computations presented above. A process v which is synchronized, executes a rule. That is, v exchanges its labels with its neighbor, checks if its labels and those of its neighbors verify one precondition among the rules composing the algorithm. If so, it updates its labels according to the relabeling part of the found rule. Then, the synchronization is broken and v and its neighbor can retry new synchronizations.

This tool can be used to visualize and experiment distributed algorithms, and therefore helps in their design and their validation. Several distributed algorithms have been already implemented and can be directly animated. However, it has been assumed that components of such a system do not fail.

Chapter 2

Self-stabilization and Local Computations

SElf-stabilization was introduced by *E.Dijkstra* [Dij74] as the property of a distributed system to recover by itself to reach one of its desired configurations in a finite number of steps, regardless of the initial system configuration. Consequently, in the context of self-stabilization, *all* configurations are *initial* configurations.

As a desirable feature of a computation in distributed systems is fault-tolerance. Specifically, transient error may be caused by message corruption, sensor malfunctioning, incorrect read/write that transforms configuration of a system to illegal configuration. That is, transient failures can potentially put the system into an illegal configuration which may continue indefinitely unless external measures are taken including the restart of the computation for example. Self-stabilization guarantees that a system can recover to reach a legal configuration in a finite number of steps. That is, every execution of a self-stabilizing algorithm reaches a "correct" configuration. Thus, self-stabilization becomes a particularly suitable approach to deal with fault-tolerance [Sch93, Dol00, Tel00]. This is one of its advantages. More advantages include robustness for dynamic topologies changes and straightforward initialization.

As locality is an important feature of distributed computing, it is important to understand how to carry on local computations without a specified initial states. In this chapter, we consider the problem of designing self-stabilizing algorithms using only local knowledge. In the other hand, we formulate the properties of these algorithms using those of rewriting systems yielding a simple and formal proofs [LMS99]. A self-stabilizing system guarantees that it will eventually reach a legal configuration when started from an arbitrary initial configuration. This behavior is called *Convergence*. After the reach of a legal configuration, the system generates only legal configurations for the rest of the execution. This behavior is called *Closure*. Most interests are given to the study of these properties according to two aspects:

1. Locality: Convergence will occur with a short distance from the illegal configuration.
2. Stabilization time: Convergence will occur after a short time.

The developed formal framework [HM05a, HM06b] allows to design and prove such algorithms. We introduce correction rules in order to self repair the possible illegal states. Such rules have higher priorities than the rules of the main algorithm which ensure that the illegal

states are repaired before continuing on the computation. Of course, we deal only with predefined illegal local configurations. Further, we have presented some techniques to transform an algorithm in a self-stabilizing one using local correction rules. Various examples are given to illustrate this method. Particularly, the problem of resolving conflicts is studied in more details. A we will see, a complete analysis is presented when its required synchronizations are achieved using randomized local election procedures.

The rest of this chapter is organized as follows. In Section 2.1, we describe the token circulation algorithm as presented in the seminal paper by E.W. Dijkstra [Dij74]. Section 2.2 surveys the various facets of stabilization. Then, in section 2.3 we give the model of computing, the definitions and terminology that we will use. Sections 2.4 and 2.5 are the heart of the chapter. In the first one, we present our formalization of the illegal configurations. In the second one, we deal with local stabilization in the local computations model. Section 2.6 describes the methodology proposed to design local self-stabilizing distributed algorithms. Such a technique is illustrated with some algorithms including SSP [SSP85], enumeration protocol [Maz97]. Then, in Section 2.7 we study the problem of resolving conflicts in the presence of transient failures. The proposed solution is implemented using synchronization based on randomized local elections and then analyzed. Section 2.8 presents an optimal schedule of the resolving conflicts algorithm for free triangle graphs. Finally, section 2.9 recaps our findings and presents some possible extensions.

2.1 Dijkstra Stabilization

Now we recall the Dijkstra's toking ring algorithm. For the first time in [Dij74], the notion of self-stabilization was introduced using a state machine model. This novel notion is studied through an algorithm devoted to deal with *mutual exclusion* problem in a directed ring.

Let $v_0, \cdots, v_{N-1}$ be a set of N processes which dispute the critical section. Here a process that can enter the critical section is named "privileged". Any algorithm which solves this problem has to ensure the following properties :

- in every configuration at most one process is privileged,
- if a process u asks to have an access to the critical section, u must end by obtaining this access. It means that no process remains privileged for ever and u will eventually become privileged.

The Dijkstra's algorithm encodes the token circulation in a logical oriented ring with an elected process called the *Top*, denoted in the following by v_0. Each process is labeled with label St to denote its state during the execution of the algorithm. Such a label is defined under an alphabet $\Sigma_D = \{0, \cdots, K-1\}$. Each process can read only the state of its predecessor and changes its own state. The function $Pred(v)$ gives the predecessor neighbor of v. Note, however, that this function implements only the network's orientation.

As stated previously, the acquiring and the loss of privilege follows:

- v_0 is privileged if $St(v_0) = St(Pred(v_0))$.
- v_i where $0 < i < N$ is privileged if $St(v_i) \neq St(Pred(v_i))$,

That is, we distinguish between two rules:

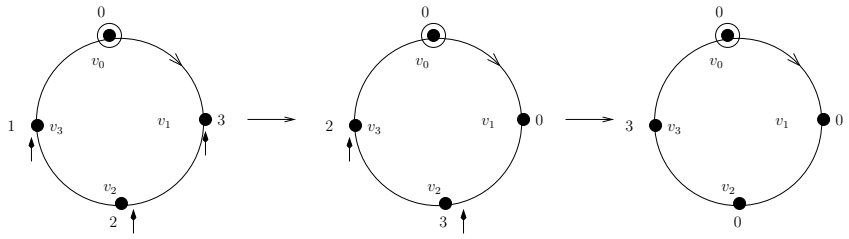


Figure 3: Example of a Dijkstra's token ring.

1. The Top process:
   If $St(Top) = St(Pred(Top))$ **Then**
   $St(Top) := St(Top) + 1 \, mod \, K$

2. The non Top process v :
   If $St(v) \neq St(Pred(v))$ **Then**
   $St(v) := St(Pred(v))$

Each privileged process is allowed to change its state, causing the loss of its privilege and thus moving the system in a new configuration. The presented algorithm *stabilizes* as we shall see for $K \geq N$. That is, starting from any configuration, each execution reaches a configuration where *mutual exclusion* is satisfied.

Let us illustrate this algorithm for a system composed of 4 machines and $K = 4$ states for each one (see Figure 3). The *Top* machine is the one in the circle. The left part of the figure is considered as the initial configuration. Here, an "enabled" process [1] is shown by a bold $\uparrow$. It is easy to see that the algorithm stabilizes as illustrated in the middle and in the right parts of the figure.

Theorem 2.1.1 *If* $K \geq N$, *then Dijkstra's token ring algorithm stabilizes.*

The proof may be found in [Tel00].

2.2 Related Works

In this section, we focus our attention on various facets of stabilization.

As stated before, self-stabilization is an attractive approach to deal with fault-tolerance in distributed computing. Originally proposed by *Dijkstra* for oriented ring topology with a distinguished node [Dij74, Dol00, Tel00], stabilization was later generalized to other problems and to arbitrary graphs. Several algorithms exist in the self-stabilization literature [APSV91, AKM+93, AS96, PV00, CDPV01]. They allow to solve very general problems that appear frequently as sub-tasks in the design of distributed systems. These elementary tasks include broadcasting of information, election of a leader, mutual exclusion, spanning tree construction, etc.

Particularly, many self-stabilizing algorithms have been already designed to deal with the problem of *depth-first token circulation* and the *propagation of information with feedback (PIF)* [Cha82]. For the first one, the protocols presented in [PV99, PV00] are devoted to trees and

[1] A process has privilege.

those of [DJPV98, Pet01] are for arbitrary graphs. For the second one, in [BDPV99] an optimal state snap-stabilizing PIF in tree networks is proposed. In [CDPV01], the authors propose a self-stabilizing PIF algorithm for arbitrary network and in [CDPV02] a snap-Stabilizing version of this algorithm is studied. Note that snap-stabilization is related to the stabilization time optimality.

However, most of these works propose global solutions which require to involve the entire system. As networks grow fast, detecting and correcting errors globally is no longer feasible. The solutions that deal locally with detection and correction are rather essential because they are scalable and can be deployed even for large and evolving networks. A few general approaches providing local solutions to self-stabilization have been proposed in [APSV91, AS96, AD97, AKY97, BH02]. These approaches study the possible refinements of self-stabilization. Hence, stabilizing systems may be refined to be fault-containing, so that the errors are corrected with cost proportional to their context [GGHP96, GG96, HH04].

Since it is expensive to design self-stabilizing algorithms, we can reduce the system requirements. To overcome this weakness, in [Dol00] the author proposed the pseudo-stabilization.

Most researches use the state model [Dij74] to study self-stabilization. In such a model, a process has access instantaneously to the states of its neighbors. What happens in the models with explicit communications like shared memory and message passing models ? There has been a general belief that stabilization and termination are not-satisfiable in the asynchronous message passing model [GM91, Dol00, AN01]. In these models, "termination" is formalized in various ways and a "timeout" mechanism is used to avoid lockout. In a recent initiative, [S.G00] proposes the use of *mobile agent* to deal with self-stabilization. Few works use such a model of computing as [S.G01, BHS01, HM01].

While many self-stabilizing protocols have been designed, still few works propose general techniques for self-stabilization. In [KP93], they showed how to compile an arbitrary asynchronous protocol into a self-stabilizing one. Unfortunately, such a transformation is expensive and the resulted protocol involves an important extra cost in the number of messages transmissions. In [APSV91], techniques to transform any *locally check-able* protocol into a stabilizing one is given. In [BBFM99], a new rewrite method for proving convergence of self-stabilizing systems was proposed. However, such an approach is applied only for protocols that are locally check-able. By contrast, [Var00] proposed a technique called *counter flushing* applied to more general set of protocols.

Fortunately, we can refer the reader to several resources which will compensate our omissions. Fine surveys of the self-stabilization area have been proposed in [Sch93, Dol00, Tel00]. Unfortunately, if the system cannot tolerate time required for stabilization self-stabilization does not help fault-tolerance. Another problem is the maximum allowed critical steps needed before system is in a legal state.

2.3 The System Model

A distributed system is modeled by a graph $G = (V, E)$, where V is a set of nodes and E is the set of edges. Nodes represent processes and edges represent bidirectional communication links. Processes communicate and synchronize by sending and receiving messages through the links. There is no assumption about the relative speed of processes or message transfer delay, the networks are *asynchronous*. The topology is unknown and each node communicates

only with its neighbors. The links are reliable and the process can fail and recover in a finite time. The failures that are tolerated in such a system are the transient failures of processes.

To encode distributed algorithms we use local computations model as presented in Section 1.2. In this model, the local state of a processor (resp. link) is encoded by the labels attached to the corresponding vertex (resp. edge). The computation in such a model is simulated by a sequence of rewritings. This model is asynchronous since several relabeling steps may be applied at the same time but we do not require that all of them have to be performed. At each step of the computation, labels are modified for a given ball composed of some node and the set of its neighbors. The modification is given according to rules depending only on the labels of nodes composing this ball. Two sequential relabeling steps can be applied in any order or even concurrently on disjoint sub-graphs.

Here we use relabeling on nodes and edges in a ball of radius 1. That is, each node in this ball may change its label and the label of its edges according to rules depending only on its own label, the label of its neighbor and the label of its corresponding edges. We assume that each node distinguishes its neighbors and knows their labels. We use the following notations:

- $L(u)$: the labels of node u,
- $L(u, v)$: the labels of the edge connecting the node u and the node v,
- $B(u)$: the ball centered on u of radius 1 which is an input data.

Abstractly, let S be a distributed system and let C be a component of S. Then,

Definitions 2.3.1

- *The local state of C, called the local configuration of the component, is composed of the complete state knowledge of C,*
- *The global state of S, called the global configuration of the system, is the list of the local states of all the components in the system S,*
- *A set of legal configurations is a subset of all the possible configurations of S. This is a set of desired configurations in which the system S is supposed to be accorded to the design goals.*

Thus, in our framework local computations we deal with:

Definitions 2.3.2

- *We denote by (G, L_i) a configuration of a distributed computation at some stage of computation i,*
- *A local configuration is then denoted by $(B_l(u), L_i)$ which is the restriction of L_i to the nodes composing the ball centered on u with radius l,*
- *A configuration (G, L') is reachable from the configuration (G, L) during the execution of the algorithm $\mathcal{R}$ iff $\exists\, p \geq 0$ such that $L \xrightarrow[\mathcal{R},p]{} L'$.*

2.4 Graph Relabeling System with Illegitimate Configurations

Local configurations will be defined on balls of radius 1 (the corresponding node and the set of its neighbors). A *star-graph* is a rooted tree where all nodes except perhaps the root have degree 1. The root will be called the center of the star-graph. Since any ball of radius 1 is isomorphic to a star-graph, illegitimate configurations will be described through their supports (the labeled star-graphs). More precisely, an illegal configuration f is a labeled star-graph, say (B_f, L_f), where B_f is a star-graph and L_f a labeling function defined on it. Sometimes, it is useful to express such a configuration by a predicate on the edges, nodes and labels of the corresponding star-graph. For instance, a graph consisting of two nodes, u labeled A and v labeled B which are connected by an edge labeled C will be written:

$$L(v) = A \; and \; \exists \; u \in B(v) : L(u, v) = C \; and \; L(u) = B$$

Definition 2.4.1 *The term "illegal" is related to some specification expressed using a set of predicates on the properties satisfied by the system states. We say that the configuration is legal iff it verifies the specifications. Then, the global configuration of a system S is legal iff the corresponding local state of each component C in the system S is legal. For a set of desired labeled graphs (G, L), denoted $\mathcal{G}_l$, we say that a local configuration $f = (B_f, L_f)$ is illegal for $\mathcal{G}_\mathcal{L}$ if for all $(G, L) \in \mathcal{G}_L$ there is no sub-graph in (G, L) which is isomorphic to f. In other words, there is no ball (neither sub-ball) of radius 1 in G which has the same labeling as f. This will be denoted by $\mathcal{G}_L \nvdash f$.*

Moreover, if $\mathcal{F}$ is a set of illegal configurations, we extend the latter notations to $\mathcal{G}_L \nvdash \mathcal{F}$ meaning that each element of $\mathcal{F}$ is an illegal configuration. A graph relabeling system with illegitimate configuration is a quadruple $\mathcal{R} = (\Sigma, \mathcal{I}, \mathcal{P}, \mathcal{F})$ where Σ is a set of labels, $\mathcal{I}$ is a subset of Σ called the set of initial labels, $\mathcal{P}$ is a finite set of relabeling rules and $\mathcal{F}$ is a set of illegitimate configurations. Let us give two examples of illegitimate configurations. Consider the following graph relabeling system given to encode a distributed spanning tree.

Assume that a unique given process is in an "active" state (encoded by the label **A**), all other processes being in some "neutral" state (label **N**) and that all links are in some "passive" state (label **0**). The tree initially contains the unique active vertex. At any step of the computation, an active vertex may activate one of its neutral neighbors and mark the corresponding link which gets the new label **1**. This computation stops as soon as all the processes have been activated. The spanning tree is then obtained by considering all the links with label **1**.

An elementary step in this computation may be depicted as a *relabeling step* by means of the relabeling rule RSPAN1, given in the following, which describes the corresponding label modifications (remember that labels describe process states):

RSPAN 1 : **The spanning rule**

Precondition :

- $L(v) = A$
- $\exists \; u \in B(v), L(u) = N$

Relabeling :

- $L(u) := A$
- $L(v, u) := 1$

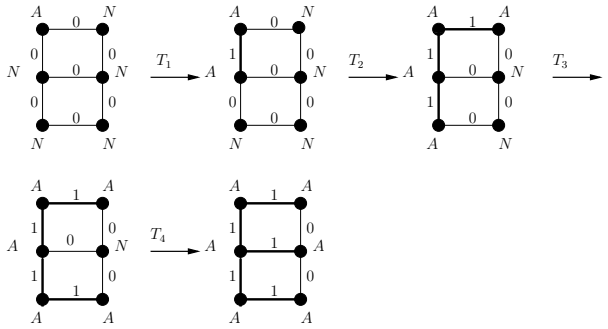


Figure 4: Example of a computation of a distributed spanning tree

Figure 5: Examples of Illegitimate configurations

Whenever an A-labeled node is linked by a 0-labeled edge to an N-labeled node, then the corresponding sub-graph may rewrite itself according to the rule.

A sample computation using this rule is given in Figure 4. Relabeling steps *may* occur concurrently on disjoint parts on the graph. When the graph is irreducible, i.e no rule can be applied, a spanning tree, consisting of the edges labeled 1, is computed.

The previous algorithm can be encoded by the relabeling system $\mathcal{R}_1 = (\Sigma_1, \mathcal{I}_1, \mathcal{P}_1)$ defined by $\Sigma_1 = \{N, A, 0, 1\}$, $\mathcal{I}_1 = \{N, A, 0\}$ and $\mathcal{P}_1 = \{RSPAN1\}$.

Clearly, we distinguish two illegitimate configurations. On the one hand, an edge labeled 1 must be incident to two nodes labeled A, as shown in the graph on the left of Figure 5. On the other hand, a node labeled A cannot be incident only to edges labeled 0, as given in the right graph. Formally, we deal with the following set $\mathcal{F}_{s_1} = \{f_1, f_2\}$ where f_1 and f_2 are:

1. $f_1 : L(v) = N \; and \; \exists \; u \in B(v) : L(v,u) = 1$
2. $f_2 : L(v) = A \; and \; \forall \; u \in B(v) : L(v,u) = 0$

2.5 Local Stabilizing Graph Relabeling Systems (LSGRS)

In this section we propose our formalization of self-stabilization using means of local computations and particularly we study their properties using those of rewriting systems yielding simple and formal proofs [LMS99]. On the other hand, as locality is an important feature of distributed computing, it is important to understand how to carry on local computations without a specified initial states. Before describing such properties, we first define the stabilization and what means stabilization time in our model.

Definition 2.5.1

- *Let $\mathcal{CRS}$ be a set of configurations. An algorithm $\mathcal{R}$ stabilizes to reach some configuration in $\mathcal{CRS}$ if the following holds: In each relabeling chain induced by the execution of $\mathcal{R}$, there is a finite suffix after which all the configurations reached are in $\mathcal{CRS}$.*

- *The stabilization time is the length of the relabeling sequence corresponding to this suffix. So a measure of the stabilization time complexity of a distributed algorithm, or of a graph relabeling system, will be the stabilization time of the longest suffix of the algorithm.*

Fact 2.5.2 *Let $\mathcal{CRS}$ be a set of configurations. Starting from any configuration in $\mathcal{CRS}$ an algorithm $\mathcal{R}$ stabilizes to $\mathcal{CRS}$ in 0 time.*

Let $\mathcal{G}_L$ be a set of configurations corresponding to the execution of an algorithm $\mathcal{R}$ on a system S. Let $\mathcal{G}_{\mathcal{LD}}$ be a set of desired configurations such that $\mathcal{G}_{LD} \subseteq \mathcal{G}_L$. An algorithm $\mathcal{R}$ is self-stabilizing for $\mathcal{G}_L$ iff it satisfies the two following properties:

Property 2.5.3

1. *Closure:* $\forall\, (G, L) \in \mathcal{G}_{LD}$, $\forall\, (G, L')$ *such that* $(G, L) \xrightarrow[\mathcal{R},*]{} (G, L')$, $(G, L') \in \mathcal{G}_{LD}$.
2. *Convergence:* $\forall\, (G, L) \in \mathcal{G}_L$, $\exists$ *an integer* $p \geq 0$ *such that* $(G, L) \xrightarrow[\mathcal{R},p]{} (G, L')$ *and* $(G, L') \in \mathcal{G}_{LD}$.

As for self-stabilizing algorithms, the closure property stipulates that a computation beginning from a configuration without illegal configurations remains without such configurations during the whole execution. The convergence however provides the ability of the relabeling system to recover automatically within a finite time (finite sequence of relabeling steps) to reach a configuration without illegal configurations.

In practice, the "stabilization phase" may involve the entire components of the system. For large systems, it is suitable to reduce as much as possible the components participating to this phase. In other words, it is interesting to involve only components nearest to the component subject to illegal local configuration. This is the goal of the local stabilization design.

A local stabilizing graph relabeling system is a triple $\mathcal{R} = (\Sigma, \mathcal{P}, \mathcal{F})$ where Σ is a set of labels, $\mathcal{P}$ a finite set of relabeling rules and $\mathcal{F}$ is a set of illegitimate local configurations.

As we shall see, the set of relabeling rules $\mathcal{P}$ is composed by the set of relabeling rules P used for the computation and some correction rules P_c. The rules of the set P_c are introduced in order to eliminate the illegitimate configurations. The latter rules have higher priority than the former in order to eliminate faults before continuing computation.

Theorem 2.5.4 *If $\mathcal{R} = (\Sigma, \mathcal{I}, \mathcal{P}, \mathcal{F})$ is a graph relabeling system with illegitimate configurations (GRSIC) then it can be transformed into an equivalent local stabilizing graph relabeling system (LSGRS) $\mathcal{R}_s = (\Sigma_s, \mathcal{P}_s, \mathcal{F}_s)$.*

Proof. We will show how to construct $\mathcal{R}_s = (\Sigma_s, \mathcal{P}_s, \mathcal{F}_s)$. It is a relabeling system with priorities. To each illegitimate local configuration $(B_f, L_f) \in \mathcal{F}_s$, we add to the set of relabeling rules the rule $Rc = (B_f, L_f, L_i)$ where L_i is a relabeling function associating an initial label with each node and edge of B_f. This relabeling function depends on the application; for example, the initial value of a node label is N in general, and the label of an edge is 0. The rule Rc is, in fact, a correction rule. Thus the set $\mathcal{P}_s$ consists of the set $\mathcal{P}$ to which is added the set of all correction rules (one rule for each illegitimate configuration). Finally, we give a higher priority to the correction rules than those of $\mathcal{P}$, in order to correct the configurations before applying the rules of the main algorithm. It remains to prove that it is a stabilizing system. The proof is equivalent to the proof of the two following lemmas.

Lemma 2.5.5 *The system $\mathcal{R}_s = (\Sigma_s, \mathcal{P}_s, \mathcal{F}_s)$ satisfies the closure property.*

Proof. Let $(G, L) \not\vdash \mathcal{F}$. If (G, L') is an irreducible graph obtained from (G, L) by applying only the rule of $\mathcal{P}$, then (G, L') does not contain an illegitimate configuration. This can be shown by induction on the sequences of relabeling steps [LMS99, MMS02]. □

Lemma 2.5.6 *The system $\mathcal{R}_s = (\Sigma_s, \mathcal{P}_s, \mathcal{F}_s)$ satisfies the convergence property.*

Proof. Let $\mathcal{G}_L$ be the set of labeled graphs (G, L) and $h : \mathcal{G}_L \longrightarrow \mathbb{N}$ be an application associating with each labeled graph (G, L), the number of its illegitimate configurations, then for a graph (G, L), we have the following properties:

- The application of a correction rule decreases $h(G, L)$.
- The application of a rule in $\mathcal{P}$ does not increase $h(G, L)$.

Since, the correction rules have higher priority than the rules in $\mathcal{P}$, and since the function h is decreasing, then it will reach 0 after a finite number of relabeling steps. Note that the last property of convergence can also be proved by using the fact that the relabeling system induced by the correction rules is noetherian. □

Let us note that the correction rules depend on the application. While the proofs above are based on the local reset (to the initial state) which can be heavy because it may induce a global reset by erasing all the computations, it is more efficient for particular applications to choose suitable corrections as we shall see in the sequel.

2.5.1 Spanning Tree Computations

We present in the sequel a spanning tree computation encoded with a local stabilizing relabeling system. We start by defining some illegitimate configurations to construct a set $\mathcal{F}_s$, then we improve the system by adding the correction rules to detect and eliminate these configurations. For the present system, we deal with the set $\mathcal{F}_{s_1}$ defined above. Thus, the correction rules are deduced from the previous configuration as follows:

RS_SPAN1 : **The correction rule 1**

Precondition :
- $L(v_0) = A$
- $\exists\, v_i \in B(v_0), L(v_i) = N, L(v_0, v_i) = 1$

Relabeling :
- $L(v_0, v_i) := 0$

RS_SPAN2 : **The correction rule 2**

Precondition :
- $L(v_0) = A$
- $\exists\, v_i \in B(v_0), L(v_i) = A, L(v_0, v_i) = 0$
- $\forall\, v_j \neq v_i \in B(v_0), L(v_0, v_j) = 0$

Relabeling :
- $L(v_0, v_i) := 1$

We assume in this system the existence of a distinguished node which is initially labeled A and which is usually correct. We define the relabeling system $\mathcal{R}_{s_1} = (\Sigma_{s_1}, \mathcal{P}_{s_1}, \mathcal{F}_{s_1})$, where $\Sigma_{s_1} = \{N, A, 0, 1\}$ and $\mathcal{P}_{s_1} = \{RSPAN1, RS\_SPAN1, RS\_SPAN2\}$ such that $RS\_SPAN1, RS\_SPAN2 \succ RSPAN1$. We now state the main result.

Theorem 2.5.7 *The relabeling system $\mathcal{R}_{s_1}$ is local stabilizing. It encodes a self-stabilizing distributed algorithm to compute a spanning tree.*

Proof. The proof of local stabilizing results from Theorem 2.5.4. To show that the result is a spanning tree, we use the following invariants which can be proved by induction on the size of the relabeling sequences:

(I1) All edges incident to an N-labeled node have labels 0.

(I2) Each edge which is labeled 1 is incident to two nodes labeled A.

(I3) The sub-graph induced by the edges labeled 1 is a tree.

(I4) The obtained tree of the irreducible graph is a spanning tree.

2.6 Automatic Generation of Local Stabilizing Algorithms

Here, we apply the developed framework to deal with self-stabilization. The method is based on two phases. The first one consists of defining the set of illegitimate configurations (GRSIC). The second phase allows to construct some local correction rules to eliminate the illegitimate configurations. Then the graph relabeling system composed of the initial graph rewriting system improved with the correction rules is a self-stabilizing system (LSGRS). We illustrate our approach by various self-stabilizing algorithms.

2.6.1 The SSP's Algorithm

In this section we present the local stabilizing SSP's algorithm [SSP85]. We consider a distributed algorithm which terminates when all processes reach their local termination conditions. Each process is able to determine only its own termination condition. The SSP's algorithm detects an instant in which the entire computation is achieved.

Let G be a graph. Each node v is associated with a predicate $p(v)$ and an integer $a(v)$. Initially $p(v)$ is false and $a(v)$ is equal to -1. Transformations of the value of $a(v)$ are defined by the following rules.

Each local computation updates the integer $a(v_0)$ associated with the vertex v_0; the new value of $a(v_0)$ depends on values associated with the neighbors of v_0. More precisely, let v_0 be a vertex and let $\{v_1, \cdots, v_d\}$ be the set of vertices adjacent to v_0.

- If $p(v_0) = false$ then $a(v_0) = -1$;
- if $p(v_0) = true$ then $a(v_0) = 1 + Min\{a(v_k) \mid 0 \leq k \leq d\}$.

We consider in this section the following assumption. For each node v, the value $p(v)$ eventually becomes true and remains true forever.

We define a relabeling function L associating with each vertex v the value $L(v) = (p(v), a(v))$. The SSP's algorithm can be encoded by a relabeling system which rules are :

RSSP1 : **The SSP rule**

Precondition :

- $p(v_0) = true$
- $a(v_0) - 1 \neq Min\{a(v_i) | v_i \in B(v_0)\}$.

Relabeling :

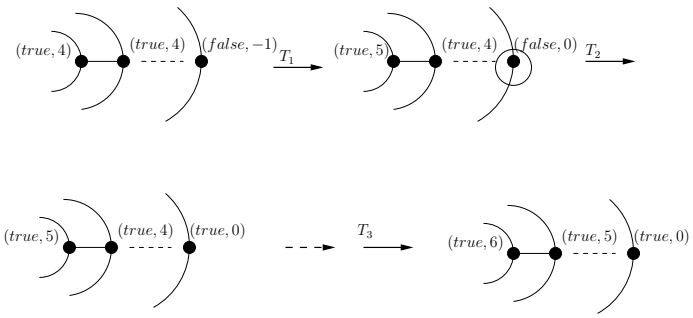


Figure 6: Example of SSP

- $a(v_0) := 1 + Min\{a(v_i)|v_i \in B(v_0)\}$.

Let us define some illegitimate configurations to construct a set $\mathcal{F}_{s_2}$, then we improve the system by adding some correction rules to detect and eliminate these configurations. The predicate $p(v)$ is maintained by an underlying protocol and each given value is correct. Hence, we assume that the predicate p is never false. Consider the set $\mathcal{F}_{s_2} = \{f_1\}$ where f_1 is: $f_1 : a(v) \geq 0 \; and \; p(v) = false$

The correction rule is defined as follows:

RS_SSP1 : **The correction rule 1**

Precondition :

- $a(v) \geq 0$
- $p(v_0) = false$

Relabeling :

- $a(v_0) := -1$.

We define the relabeling system $\mathcal{R}_{s_2} = (\Sigma_{s_2}, \mathcal{P}_{s_2}, \mathcal{F}_{s_2})$, where $\Sigma_{s_2} = \{\{false, true\} \times [-1, diam(G)]\}$, and $\mathcal{P}_{s_2} = \{RSSP1, RS\_SSP1\}$ where $RS\_SSP1 \succ RSSP1$.

If we consider Figure 6, assume that during the execution of the SSP's algorithm, some node (see the node in the circle) has changed incorrectly its state from $(false, -1)$ to $(false, 0)$ (after step T_1). In this case, this node is in the illegal configuration f_1, and the rule RS_SSP1 will be applied as it has higher priority than the main rule (step T_2). Note that the correction is done immediately and locally. We now state the main result:

Theorem 2.6.1 *The relabeling system $\mathcal{R}_{s_2}$ is local stabilizing. It encodes a self-stabilizing SSP's algorithm.*

Proof. The proof of local stabilizing results from Theorem 2.5.4. To show that the result is correct, we use the following invariants which can be proved by induction on the size of the relabeling sequences:

(I1) If $p(v_0) = false$, then $a(v_0) = -1$

(I2) If $p(v_0) = true$, then $a(v_0) := 1 + Min\{a(v_i)|v_i \in B(v_0)\}$.

□

2.6.2 Enumeration Protocol

An enumeration algorithm on a graph $G = (V, E)$ is a distributed algorithm such that the result of any computation is a labeling of the nodes that is a bijection from V_G to $1, 2, \cdots, \#V$. First, we give a description of the initial enumeration algorithm [Maz97]. Every node attempts to get its own name, which shall be an integer between 1 and $\#V$.

The Mazurkiewicz's Enumeration Algorithm. A node chooses a name and broadcasts it with its neighbor-hood (i.e. the list of the name of its neighbors) all over the network. If a node u discovers the existence of another node v with the same name, then it compares its local view, i.e. the labeled ball of center u, with the local view of its rival v. If the local view of v is "Stronger", then u chooses another name. Each new name is broadcasted with the local view again over the network. At the end of the computation it is not guaranteed that every node has a unique name, unless the graph is non ambiguous [Maz97]. However, all nodes with the same name will have the same local view.

The crucial property of the algorithm is based on a total order on local views such that the "Strength" of the local view of any node cannot decrease during the computation. To describe this local view we use the following notation: If v has degree d and its neighbors have names $n_1, n_2, \cdots n_d$ with $n_1 \geq \cdots \geq n_d$, then $LV(v)$, the local view, is the d-$tuple$ $(n_1, n_2, \cdots n_d)$. Let $\mathcal{LV}$ be the set of such ordered tuples. The alphabetic order defines a total order $\preceq$ on $\mathcal{LV}$. The nodes v are labeled by triples of the form (n, LV, GV) representing during the computation:

- $n(v) \in \mathbb{N}$ is the name of the node v,
- $LV(v) \in \mathcal{LV}$ is the latest view of v,
- $GV(v) \subset \mathbb{N} \times \mathcal{LV}$ is the mailbox of v and contains all the information received at this step of the computation. We call this set the global view of v.

The initial labels of each node are $(0, \phi, \phi)$ and as presented in the previous, the algorithm is composed of two rules: (1) transmitting rule, and (2) renaming rule.

A Local Stabilizing Enumeration Algorithm. We briefly present in the sequel a new enumeration algorithm encoded by local stabilizing relabeling systems. This protocol is optimal compared to the version presented in [God02].

In a correct behavior, when a name of v_0 is already chosen by another node v_i, v_0 (resp. v_i) will receive this information and change its name if the local view of v_0 (resp. v_i) contains the older modifications.

In a corrupted behavior, when a name of v_0 is corrupted, it detects this corruption, changes its name to -1 and initializes its states, then one of its neighbors v_i detects this change, corrects the state of v_0. After that, v_0 chooses another number to rename itself.

We start by defining some illegitimate configurations to construct a set $\mathcal{F}$, then we improve the system by adding the correction rules to detect and to eliminate these configurations. The node v_0 is said to be corrupted or in the illegitimate configuration, if one of its components is changed using extra relabeling. This relabeling does not correspond to those of the previous rules. We can define the following kind of corrupted behaviors:

1. Corruption of the name,

2. Corruption of the local view,

3. Corruption of the global view.

Such configurations are expressed using predicates in order to build the set of illegitimate configurations. Therefore, a set of correction rules are added to the system to encode the local detection and correction of these configurations.

This protocol is easy to understand and its translation from the initial algorithm requires little changes. The proof is decomposed into two steps. First the proof of self-stabilization which is based on our developed framework. Second the proof that this protocol does its expected task which is based on the same as [Maz97]. For the complexity study, we show that our protocol is better than [God02]. In this work, we had also shown that self-stabilization meets global detection of termination such as [AN01]. For an in-depth description of this protocol and its proof of correctness, we refer the reader to the original papers [Maz97, God02] and to our contributions [HM06b, HM06a].

2.7 Self-stabilizing Distributed Resolving of Conflicts

In this section, we investigate the problem of resolving conflicts in a distributed environment in the presence of transient failures using only local knowledge. The stabilization time is computed in terms of computation steps as the previous section, then approximated according to the needed synchronizations.

Several papers investigate the mutual exclusion problem. In [Lam74], the author has given a simple solution to this problem in spite of failures in a complete graph. That is, a validating and checking approach. The probabilistic (or randomized) approach had been introduced in [LR81] for resolving the dining philosophers problem. This algorithm is the base of many studies such that of [DFP02]. Randomization is used to cope the case of asynchronous and anonymous networks [Her90, KY97]. The work closest to our is [MSZ06]. The main difference between our work and [MSZ06] is: Here we present a self-stabilizing solution.

Resolving conflicts problems in distributed systems has been active more than three decades. In fact, when a group of processors, composing the system, needs access or uses some resource that cannot be used simultaneously by more than one processor, for example some output device, the system must ensure some properties to manage such conflicts between processors. Here, we study such a problem in distributed environment. Resources subject to conflicts are called *critical resources* and by extension the use of such resources is named *critical section*.

We consider the case when a process needs to have access to all its resources to do any computation. This problem is named *drinking philosophers problem* and was introduced by Chandy and Misra in [CM84] as a generalization of the *dining philosophers problem* [Dij72], one of the famous paradigms in distributed computing. In a generalization of the drinking philosophers problem, a philosopher can choose one or many bottles and not only all of them [LW93, BBF01]. In this paper, we consider a network of processes sharing a set of resources. The resources are placed on the edges of the underlying graph. Any algorithm which solves this problem has to ensure the following properties :

- no shared resource is accessed by two processes at the same time, that is, the algorithm ensures the *mutual exclusion* on shared resources,

- if two processes p_1 and p_2 do not share a resource, and hence are not adjacent in the underlying graph, then they can access their resources independently, and possibly at the same time, that is, the algorithm ensures the *concurrency* property,
- if a resource is asked by two processes p_1 and p_2 and if p_1 formulates its request before p_2 then it must enter the resource before p_2, this is called the *ordering* property,
- if a process p asks to have an access to all the resources it needs, p must end by obtaining this access, this is called *liveness* property.

In our graph the resources are placed on the edges, node u is a neighbor of node v if u and v potentially share a resource. A process can perform its computation if all its needed resources are allocated to it. Each node v is labeled $L(v) = (St(v), Ord(v))$, where $St(v)$ denotes the current state of v : *tranquil*, *thirsty* or *drinking*. These states will be respectively encoded by the labels T, Th or D. The label $Ord(v)$ is an integer to manage the order of process requests.

For a sake of clarity and in order to make easier the comparison of our work with some relevant literature, we use the term abnormal instead of illegitimate.

Description of the Algorithm in a Normal Behavior [MSZ06]. Initially all the vertices of the graph are tranquil (this is encoded by the label $(T, -1)$). At each step of the computation, an $(T, -1)$-labeled vertex u may ask to enter the "CS", instead of critical section, which means that it becomes thirsty. In this case u changes its label to (Th, i) where $i = Max\{Ord(v)|v \in B(u)\} + 1$. So, the order of u is the most of its neighbors. In a complete graph, this order can be seen as a universal time since only one node can change its label to Th at the same time.

If a vertex u, with a label (Th, i), has no neighbor in the critical section (labeled $(D, -1)$) and no neighbor with a label (Th, j) where $j < i$ (it has the lowest rank of its neighbors), the vertex u can enter the critical section. That is, u will have the label $(D, -1)$.

Once, the vertex in the critical section had terminated its work, it changes its label to $(T, -1)$.

2.7.1 Our Algorithm

Clearly, all the computations starting from configurations where all nodes are $(T, -1)$ using a scheme as the normal behavior are without abnormal configurations. Since self-stabilization deals with computations starting from any configuration, we will consider in our design the fact that some abnormal configurations appear in the started configuration. The goal of the self-stabilization is to ensure that after a finite time, such configurations will disappear. We consider the following abnormal configurations:

- When a node u labeled $St(u) = Th$ has a neighbor v labeled $St(v) = St(u)$ and $Ord(u) = Ord(v)$. We denote this configuration by f_1.
- When a node u labeled $St(u) = D$ has a neighbor v labeled $St(v) = St(u)$. We denote this configuration by f_2.

Node satisfying the abnormal configuration f_2 is called f_2ab. Node satisfying f_1 is called f_1ab. Other nodes are called f_0ab. Let (G, L) be any configuration. Let $\mathcal{G}_{\mathcal{L}}$ be the set of labeled graphs (G, L) and $AB : \mathcal{G}_{\mathcal{L}} \times \{f_1, f_2\} \longrightarrow \mathbb{N}$ be an application associating to each labeled graph

(G, L), the number of its abnormal configurations. We write $AB(L, f)$ to denote the number of abnormal configurations f which appears in L. At each step of the computation, we consider the following cases: In the first case, $AB(L, f_1) > 0$ and $AB(L, f_2) = 0$. It means that there are f_1ab nodes and f_0ab nodes. Nodes labeled $(T, -1)$, (Th, i), $(D, -1)$ that are not f_1ab nodes behave as described in the normal behavior. Each f_1ab node labeled (Th, i) such that it has neighbor also labeled (Th, i) updates its label to (Th, j) where $j = Max\{Ord(v)|v \in B(u)\}+1$. So, the order of u is the most of its neighbors. For the second case, $AB(L, f_1) = 0$ and $AB(L, f_2) > 0$. Each of the f_0ab nodes follows the same behavior as the normal behavior. If a node u is a f_2ab node, it changes its label to (Th, i) where $i = Max\{Ord(v)|v \in B(u)\} + 1$. Finally, the third case $AB(L, f_1) > 0$ and $AB(L, f_2) > 0$. Nodes that are neither f_1ab or f_2ab nodes follow the normal behavior. Both f_1ab and f_2ab nodes change their labels to (Th, i) where $i = Max\{Ord(v)|v \in B(u)\} + 1$.

The algorithm may be encoded by the following graph relabeling system. In the sequel, it is referred as the $\mathcal{SRC}$ algorithm.

Algorithm 4 Self-stabilizing algorithm for resolving conflicts ($\mathcal{SRC}$ algorithm)

RS_CONF1 : **The node u becomes thirsty, it forms its request to enter in the "CS"**

Precondition :
- $St(u) = T$

Relabeling :
- $St(u) := Th$
- $Ord(u) := Max\{Ord(v), \forall v \in B(u)\}+ 1$

RS_CONF2 : **Node u is elected in its local ball, so it enters in the "CS"**

Precondition :
- $St(u) = Th$
- $\neg\exists v \in B(u)$ such that $St(v) = D$
- $Ord(u) = Min\{Ord(v), \forall v \in B(u)\}$

Relabeling :
- $St(u) := D$
- $Ord(u) := -1$

RS_CONF3 : **Node u terminates its use of the "CR", then it leaves the critical section**

Precondition :
- $St(u) = D$

Relabeling :
- $St(u) := T$

RS_CONF4 : **Thirsty node u finds one of its neighbor also thirsty with the same order, so it changes its order**

Precondition :
- $St(u) = Th$
- $\exists v \in B(u)$ such that $St(v) = Th$ and $Ord(u) = Ord(v)$

Relabeling :
- $Ord(u) := Max\{Ord(w), \forall w \in B(u)\} + 1$

RS_CONF5 : **Node u in the "CS" finds one of its neighbor also in the "CS"**

Precondition :
- $St(u) = D$
- $\exists v \in B(u)$ such that $St(v) = D$

Relabeling :
- $St(u) := Th$
- $Ord(u) := Max\{Ord(w), \forall w \in B(u)\} + 1$

2.7.2 Proof of Correctness and Analysis

We will prove that the $\mathcal{SRC}$ algorithm satisfies the closure and the convergence properties assuming that the correction rules (RS_CONF4, RS_CONF5) have highest priority than those of RS_CONF1, RS_CONF2, RS_CONF3. Then, we compute the stabilization time in terms of relabeling steps.

Lemma 2.7.1 *Starting from a configuration (G, λ) without abnormal configuration, all the configurations reached during an $\mathcal{SRC}$ -computation remains without abnormal configurations.*

Proof. The correctness proof is by induction on the size of the relabeling sequence. Suppose (G, L_0) is such that $AB(L_0, \mathcal{F}) = 0$. The only rules possibly applied are RS_CONF1, RS_CONF2, RS_CONF3. The application of such rules does not generate abnormal configuration. So, $\forall (G, L_1), (G, L_2), (G, L_3)$ such that $(G, L) \xrightarrow[\mathcal{SCR}]{RS\_CONF1} (G, L_1)$, $(G, L) \xrightarrow[\mathcal{SCR}]{RS\_CONF2} (G, L_2)$, $(G, L) \xrightarrow[\mathcal{SCR}]{RS\_CONF3} (G, L_3)$, are such as $AB(L_1, \mathcal{F}) = AB(L_2, \mathcal{F}) = AB(L_3, \mathcal{F}) = 0$. Thus the algorithm operates correctly in this case. Now, we suppose that the algorithm works correctly for all relabeling sequences with size $k-1$. Consider applying the algorithm on a configuration reached after any computation of size $k-1$. We denote such a configuration (G, L_{k-1}). Since $AB(L_{k-1}, \mathcal{F}) = 0$ it suffices to consider (G, L_{k-1}) as (G, L_0). In fact, according to the first case ("trivial case") the property holds. □

Lemma 2.7.2 *Starting from a configuration (G, L) with abnormal configuration, there is a finite suffix of the $\mathcal{SRC}$ -computation after which all the reached configurations are without abnormal configurations.*

Proof. The goal is to show that after a finite number of relabeling steps, the number of abnormal configurations decreases to reach 0. We will use an induction on the number of the abnormal configurations. Let (G, L) be a started configuration. The case of $AB(L, \mathcal{F}) = 0$ corresponds to the closure property. Suppose that $AB(L, \mathcal{F}) = 1$, and let node u be the abnormal node. Node u applies RS_CONF4 (resp. RS_CONF5) if it is a f_1ab node (resp. if it is a f_2ab node). So after the application of such a rule, the number of abnormal configurations in the resulting labeled graph is equal to 0. Now suppose that starting from any labeled graph (G, L_x) such that $AB(L_{x-1}, \mathcal{F}) = x-1$, after the application of a finite relabeling rules denoted by $f(x-1)$ the resulting labeled graph is without abnormal configurations. We consider an application of the algorithm on a labeled graph (G, L_x) such that $AB(L_x, \mathcal{F}) = x$. Let (H, γ) be a sub-labeled graph of (G, L) such that $AB(\gamma, \mathcal{F}) = x - 1$. From the hypothesis, after a finite number of relabeling rules, (H, γ) becomes without abnormal nodes. We denote such a configuration, in the graph H, (H, γ'). Then, it suffices to consider the "trivial" case of one abnormal node applied to the resulting labeled graph (G, L') composed of (H, γ'). Hence, the lemma holds. □

For the time complexity requirements of our algorithm we deal with the worst case of the convergence time. In other words, the cost of the algorithm is the number of steps involved by the algorithm to reach a configuration without any abnormal configuration. We use the following properties to give the stabilization time of our algorithm:

1. Each node applies one rule in the set of correction rules to correct itself.
2. The application of the correction rules does not add abnormal configurations.
3. In the worst case, for n_1 D-abnormal nodes and n_2 Th-abnormal nodes, the nodes apply $n_1 + n_2$ correction rules.

For the space requirements, each node v is labeled $L(v)$ using the three following components: (1) $B(v)$, (2) $Ord(v)$ and (3) $St(v)$. Thus, to encode the first component of a label, every node needs to know the set of its neighbors. So, every node v requires $(deg(v) \times \log deg(v))$ bits to store this component [2], where $deg(v)$ is the degree of v. For the second component,

[2] We use $\log x$ to denote $\log_2 x$.

to avoid the use of the unbounded value of the order (or ∞) we assume that an upper bound of the possible value taken by the variable Ord is known. Let $\mathcal{N}$ be such a value. Note that such a bound is used only for analysis: It is not required for the proof of correctness of the algorithm. Then, the computation of the label $Ord(v)$ will be based on this knowledge: $Ord(v) := (Max\{Ord(v)|v \in B(u)\} + 1) \ mod \ \mathcal{N}$, where mod is an abbreviation of $modulo$ defined as: Given two numbers a and b, $a \ modulo \ b$ is the remainder, on division of a by b. Note that such a knowledge is not used to prove the correctness of our algorithm. By taking into account the last component, we can claim that the space requirement is $O(deg(G) \times \log deg(G) + \log \mathcal{N})$ bits per node.

The following result shows our first contribution.

Theorem 2.7.3 *The $\mathcal{SRC}$ algorithm encodes a distributed self-stabilizing algorithm for resolving conflicts. When the $\mathcal{SRC}$ algorithm is applied on a graph $G = (V, E)$, its time complexity is in $O(\#V)$ steps and its space complexity is in $O(deg(G) \times \log deg(G) + \log \mathcal{N})$ bits per node. For some upper bound of the possible value of the order $\mathcal{N}$.*

2.7.3 Implementation Using Randomized Synchronizations

Now, we present some results related to the implementation of the $\mathcal{SRC}$ algorithm using synchronizations based on a randomized local election. We will show that, starting from any configuration, the protocol converges to satisfy its specifications and after that, it satisfies such specifications for ever.

To implement the GRS, and hence the algorithm it encodes, we need a procedure to ensure that at any step, no two adjacent vertices apply one of the rules at the same time. To do so, we use a randomized procedure studied in [YM02] and used to implement local computations: In a round, a vertex chooses a random real value and then sends it to all its neighbors. When a vertex v receives all the messages, it compares its value to those sent by its neighbors. If v has the maximum value, then it knows that it is elected in the ball of center v and of radius 1 (and then can perform a step of the GRS). Otherwise, it keeps standing until the next round where it will try again to be locally elected, this local election is called LE_1. In [YM02], the authors show that if d is the degree of v then the expected waiting time for v to be locally elected is $d + 1$.

We show an example of a randomized procedure which implements the local election algorithm LE_1. This procedure may be used to implement the local computation Lc_1 as presented in Section 1.2.5. We refer to the ball of radius 1 as a synchronized star, or simply a star. Two results can be obtained by this function:

- starCenter : is the center vertex of the local synchronized star. It receives from its neighbors their labels and it changes its label.
- starBorder : is a vertex neighbor of a starCenter. Note that a vertex can belong to more than one synchronized stars.

Figure 7 shows an example of this synchronization. The LE_1 procedure is implemented by the following algorithm:

Before proving the correctness of the algorithm, let us recall some properties of the graph relabeling system we described in the last section.

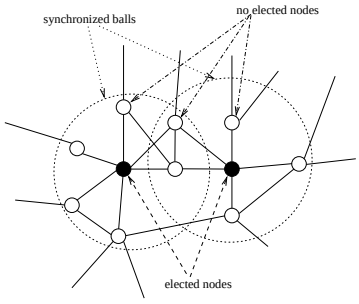


Figure 7: Example of the used local election procedure

Algorithm 5 Local synchronization procedure 1 (LE_1)

```
myNumber = RandomNumber();//Choose a random number
sendAll(myNumber);//Send my number to all my neighbours
for (door=0;door<arity;door++){
  neighborNumber[door] = receiveFrom(door);
}
maxNumber=Max(neighborNumber);//Compute the max number of the neighbors
if(myNumber>maxNumber){ //My number is the greater
  sendAll(center);
  return starCenter;
}
else
if (receive()==center){
  return starBorder;
}
```

Let (G, L) be a labeled graph. Let (G, L') be a labeled graph such that: $(G, L) \xrightarrow[\mathcal{SRC}]{*} (G, L')$. Then the graph (G, L') satisfies the following properties:

Properties 2.7.4

1. *Each node u with label $(T, -1)$, which wants to enter the critical sections (CS), changes its label to $(Th, n+1)$, where $n = Max\{Ord(v) | v \in B(u)$ and $L'(v) = (X, Ord(v)), X \in \{T, Th, D\}\}$.*

2. *To change its label to $(D, -1)$, a (Th, i)-labeled node must not have a neighbor with the label $(D, -1)$ or with the label (Th, j) where $j < i$.*

3. *Each thirsty node u with label (Th, i) connected with a node v labeled also (Th, i), updates its label to $(Th, n+1)$, where $n = Max\{Ord(v) | v \in B(u)$ and $L'(v) = (X, Ord(v)), X \in \{T, Th, D\}\}$.*

4. *Each node u supposed in the "CS" (labeled $St(u) = D$) which finds one of its neighbor v also supposed in the "CS", sets its label to $(Th, n+1)$, where $n = Max\{Ord(v) | v \in B(u)$ and $L'(v) = (X, Ord(v)), X \in \{T, Th, D\}\}$.*

Concerning the concurrency property, in [YM02], it is proved that if a vertex v is locally elected in a ball $B(v)$ then there is no other vertices locally elected in $B(v)$, meaning that v can perform its action (v can apply a rule of the GRS) without influence in the behavior of the vertices at a distance greater than 1 from v. That is :

Lemma 2.7.5 *The concurrency property is verified by the algorithm.*

To capture the worst cases of the abnormal configurations, we will use the following definition:

Definition 2.7.6 *Let $a > 0$ be an integer and let $\mathcal{AB}_a$ denote the set of abnormal vertices, including f_1ab, f_2ab vertices, in G at the a^{th} step of the algorithm.*

1. *Let $P_k = (v_0, v_1, \cdots, v_{k-1})$ be a path in $\mathcal{AB}_a$. Each of the nodes v_0 and v_{k-1} is called the end-path node. Then,*
   (a) *P_k is a* D-abnormal path *if* $\forall\, j \in [0..k-1],\ L(v_j) = (D, -1)$.
   (b) *P_k is a* Th-abnormal path *if* $\forall j \in [0..k-2]$, *if* $L(v_j) = (Th, i)$ *then* $L(v_{j+1}) = (Th, i)$.
2. *Let $\mathcal{P}$ be a set of paths P_k that covers $\mathcal{AB}_a$. In other words, $\mathcal{P}$ covers the set of all abnormal vertices.*

To prove the mutual exclusion we will be interested to study the possible construction and destruction of the *D-abnormal* configurations and for the liveness property we consider the *Th-abnormal* configurations. In both studies we consider the abnormal set of paths that covers the set of all abnormal vertices.

To eliminate a D-abnormal node v_0 it suffices that v_0 changes its label to (Th, i). That is, we focused on the expected time necessary for v_0 to change its label to (Th, i). When a vertex v is labeled $(D, -1)$, it needs to become locally elected to change its label to (Th, i). Recall that from [YM02], the expected waiting time for a vertex v to become elected in a ball $B(v)$ is

$$\mu(v) = deg(v) + 1. \tag{1}$$

Where $deg(v)$ is the degree of the vertex v.

Now, we deal with abnormal paths as Definition 3. Let $\mathcal{P}_D$ be a cover composed of D-abnormal paths, so $\mathcal{P}_D = \{P_i^1, P_j^2, \cdots P_k^l\}$. The worst case corresponds to the elimination of the abnormal paths one by one. We will study the longest D-abnormal paths because they influence the needed time to achieve the mutual exclusion property. Indeed, if $P_k = (v_0, v_1, \cdots, v_{k-1})$ is such a path, it is clear that the application of the rule RS_CONF5 by v_0 or v_{k-1} decreases the length of the path by 1. However, the application of the same rule by other nodes decreases the length by 2. The worst case corresponds to the application of the RS_CONF5 by the end-path nodes. We consider the case when $v_0, v_1, \cdots, v_{k-1}$ are elected locally in that order.

Lemma 2.7.7 *Let $P_k = (v_0, v_1, \cdots, v_{k-1})$ be a D-abnormal path of length k. If τ_{Dk} denotes the time to obtain local elections on v_0,v_1, $\cdots$ and v_{k-1} in that order then*

$$E(\tau_{Dk}) = \sum_{i=0}^{k-1} (deg(v_i) + 1). \tag{2}$$

Proof. Let t_i denotes the random variable defined by : $t_i = 1$ if v_i is locally elected, and $t_i = 0$ otherwise. It is easy to see that $\tau_{Dk} = \sum_{i=0}^{k-1} t_i$. By the linearity of the expectation, we have

$$E(\tau_{Dk}) = \sum_{i=0}^{k-1} E(t_i) = \sum_{i=0}^{k-1} \mu(v_i) \tag{3}$$

then by (1), this ends the proof. □

Lemma 2.7.8 *Eventually, after a finite time the mutual exclusion property is verified by the algorithm.*

Proof. The goal of this lemma is to show that after a finite time the computation will reach a configuration deprived of f_2ab nodes. So we are interested to study the worst case or the upper bounds. That is, the case of a $\mathcal{P}_D$. From the previous, the expected waiting time to destroy a D-abnormal path corresponds to the expected waiting time for each of its vertices to become locally elected. Thus, the expected waiting time to destroy a $\mathcal{P}_D$ cover corresponds to the expected waiting time to destroy each of its D-abnormal paths. According to (2) and (3), if τ_D denotes the time to obtain local elections on v_0, and then local elections in its D -abnormal paths P_0, $\cdots$ and $P_{deg(v_0)-1}$ in that order then we deduce:

$$E(\tau_D) \leq \#\mathcal{P}_D \sum_{i=0}^{diam(G)-1} (deg(v_i)+1). \tag{4}$$

Where $\#\mathcal{P}_D$ denotes the number of D-abnormal paths in $\mathcal{P}_D$. □

From the previous Lemmas and Properties 2, the following holds:

Theorem 2.7.9 *Eventually, after a finite time the mutual exclusion property is verified by the algorithm and then both ordering and mutual exclusion properties are verified for ever.*

To prove that the computation reaches a configuration satisfying the liveness property, we use the same reasoning as the previous. First, to eliminate all the Th-abnormal nodes it suffices that each of them changes its label to (Th, max_ord) such that its order becomes the most of its neighbors. In fact, we study the expected time necessary for a vertex v_0 to change its label to (Th, max_ord). To do this, v_0 must become locally elected to change its label (1). For the liveness property we consider Th-abnormal paths in the sense of Definition 3. In fact, if $P_k = (v_0, v_1, \cdots, v_{k-1})$ is such a path, it is clear that the application of the rule RS_CONF4 by v_0 or v_{k-1} decreases the length of the path by 1. However, the application of the same rule by other nodes decreases the length by 2. The worst case corresponds to the application of the RS_CONF4 successively by the end-path nodes. We consider the case when $v_0, v_1, \cdots, v_{k-1}$ apply the rule RS_CONF4 in that order. The proof of the following lemma is the same as the lemma used to show that the D-abnormal paths will be destroyed in a finite time.

Lemma 2.7.10 *Let $P_k = (v_0, v_1, \cdots, v_{k-1})$ be a Th-abnormal path of length k. If τ_{Thk} denotes the time to obtain local elections on v_0,v_1, $\cdots$ and v_{k-1} in that order, then*

$$E(\tau_{Thk}) = \sum_{i=0}^{k-1} (deg(v_i)+1). \tag{5}$$

The goal of the next lemma is to show that after a finite time the computation will reach a configuration deprived of f_1ab nodes. We consider the case of Th-abnormal cover denoted by $\mathcal{P}_{Th}$. Using the same reasoning of the D -abnormal configurations, the expected waiting time to destroy all the Th-abnormal nodes corresponds to the expected waiting time to destroy each of the Th-abnormal paths in $\mathcal{P}_{Th}$. According to (1) and (5), we deduce:

$$E(\tau_{Th}) \leq \#\mathcal{P}_{Th} \sum_{i=0}^{diam(G)-1} (deg(v_i)+1). \tag{6}$$

Then,

Lemma 2.7.11 *Eventually, after a finite time the liveness property is verified by the algorithm.*

After the reach of a configuration satisfying the liveness property, it remains to prove that the liveness property is satisfied for ever. We need to start by proving some lemmas verified by the GRS which encodes the algorithm.

Definition 2.7.12 *Let $a > 0$ be an integer and let $\mathcal{H}_a$ denotes the set of vertices labeled (Th, i) in G at the a^{th} step of the algorithm. Let $P_k = (v_0, v_1, \cdots, v_{k-1})$ be a path in $\mathcal{H}_a$.*
P_k is a consecutive path *if $\forall j \in [0..k-2]$, if $L(v_j) = (Th, i)$ and $L(v_{j+1}) = (Th, l)$ then $l = i+1$.*

We are interested on the investigations on the consecutive paths because they are the worst cases for our algorithm. Indeed, if $P_k = (v_0, v_1, \cdots, v_{k-1})$ is such a path, it is clear that v_{k-1} will not be allowed to change its label to $(D, -1)$ until all the vertices v_0, v_1,$\cdots$, and v_{k-2} have changed their labels in that order.

A vertex which is labeled $(T, -1)$ must become locally elected a first time to change its label to (Th, i), and a second time to verify that it has the lowest rank of its neighbors and then changes its label to $(D, -1)$. So, if at a step s, a consecutive path $P_k = (v_0, v_1, \cdots, v_{k-1})$ is formed, this means that v_0, v_1, $\cdots$, and v_{k-1} were elected in that order. Using the same reasoning, it is easy to see that the vertex v_{k-1} will change its label to $(D, -1)$ if the vertices v_0, v_1, $\cdots$, and then v_{k-1} were elected again in that order.

That is, in this section, we focus on the expected time necessary for $v = v_{k-1}$ to change its label to $(D, -1)$ since it is equal to the expected time for P_k to be constructed.

When a vertex is labeled (Th, i), it needs to become locally elected to change its label to $(D, -1)$. But, the time it will take for the vertex v_{k-1} to become able to change its label to $(D, -1)$ will be the time for the vertices $v_0, v_1, \cdots, v_{k-2}$ to become locally elected respectively. It follows the following lemma proved using the same technique as Lemma 4.

Lemma 2.7.13 *Let $P_k = (v_0, v_1, \cdots, v_{k-1})$ be a consecutive path of length k. If τ_k denotes the time to obtain local elections on v_0, v_1,... and v_{k-1} in that order, then*

$$E(\tau_k) = \sum_{i=0}^{k-1} (deg(v_i) + 1). \tag{7}$$

Corollary 2.7.14 *The upper bound of the $E(\tau_k)$ corresponds to the case of complete graph. Since*

$$2k - 2 \leq E(\tau_k) \leq 2m + k. \tag{8}$$

Where m denotes the number of edges. Let n be the number of vertices. Hence, we have

$$E(\tau_k) \leq \frac{n(n+1)}{2} + n - 1. \tag{9}$$

The aim of the next lemmas is to show that the use of a randomized procedure ensures that the probability of obtaining a *long* consecutive path is small.

Lemma 2.7.15 *Let v be a vertex and let $P = (v_0 = v, v_1, \cdots, v_{k-1})$ be a consecutive path of length k. Then with probability $1 - \frac{deg(v)+1}{\alpha}$, we have $k \leq \alpha$, $\forall \alpha > d_{max} + 1$, where d_{max} is the maximum degree of G.*

Proof. Suppose that $L(v) = (Th, i), L(v_1) = (Th, i+1), \cdots, L(v_{k-1}) = (Th, i+k-1)$. Having such a path in our graph means that there were local elections on v_0, $v_1, \ldots,$ and then v_{k-1} in that order, and v has not been locally elected during at least k rounds.

Let t be the random variable which counts the number of rounds it takes for v to be locally elected. Using the Markov inequality [Fel60] and (1), we have

$$\mathbb{P}r(t > \alpha) \leq \frac{\mu(v)}{\alpha} = \frac{deg(v)+1}{\alpha},\ \forall \alpha > deg(G)+1.$$

□

Corollary 2.7.16 *For graphs where the degree is bounded, mainly if it is in $O(\log n)$, it suffices to take $\alpha = \log n$ to obtain that with probability $1 - O(\frac{1}{\log n})$, there is no consecutive path of length more than $\log n$.*

That is we have the expected theorem:

Theorem 2.7.17 *Eventually, after a finite time the liveness property is verified by the algorithm and then it remains verified for ever.*

Remark. As mentioned above, it is possible to destroy an abnormal path in one round. Nevertheless, a straightforward computation insures that this happens with a small probability. Indeed, this probability can be upper bounded by $\frac{1}{2^{\frac{k}{2}}}$ for any path of length k.

2.8 Optimal Schedules of the Self-stabilizing Resolving of Conflicts

This section tries to answer the following question: Is the activation of a subset of the set of abnormal nodes suffices to destroy all the abnormal configurations? As mentioned above, the use of the randomized procedure allows that the probability that the stabilization follows the "scheme" of abnormal paths as shown in the previous is small. It corresponds to the upper bounds. We study the behavior of our algorithm when it is applied to some graph structure: We propose a class of graphs for which the stabilization time is optimal regardless of the number of abnormal configurations. To characterize such graphs, we use the following definitions.

Definitions 2.8.1 *Let $G = (V, E)$ be a graph. Then,*

1. *We say that G is "triangle-free" iff $\forall (u,v),(u,w) \in E, (v,w) \notin E$.*
2. *We say that a set of nodes $C(V)$, such that $C(v) \subseteq V$, is a cover of G iff the following holds: $\forall u \in V, \exists v \in C(V)$ such that $(u,v) \in E$.*
3. *Let $\mathcal{C}(\mathcal{V})$ be the set of all covers of $G = (V, E)$. The set of node $c(v)$ is a minimal-cover of G iff $\#c(v) = Min\{\#C(v),\ \forall C(v) \subseteq \mathcal{C}(\mathcal{V})\}$.*

Figure 8 gives a sample schedule starting from a configuration with 3 abnormal nodes, as shown in the dashed square. The cover of the graph is the set of nodes in the square. Therefore, to destroy all the abnormal configurations it suffices to activate the minimal-cover of the corresponding network. This means that the synchronization procedures will take into account such knowledge. In this case, the stabilization is achieved in one round.

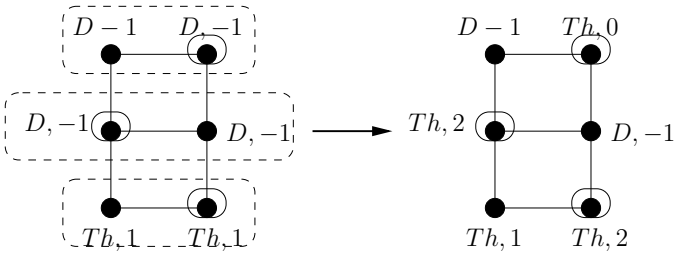


Figure 8: Example of an optimal behavior with 3 abnormal nodes

2.9 Status and Future Works

We presented a formal method to design self-stabilizing algorithms by using graph rewriting systems (GRS). This method is based on two phases. The first phase consists of defining the set of illegitimate configurations (GRSIC). The second phase allows to construct some local correction rules to eliminate the illegitimate configurations. Then the graph relabeling system composed of the initial graph rewriting system improved with the addition of the correction rules is a self-stabilizing system (LSGRS). We obtain a general approach to deal with fault-tolerance in distributed computing: The self-stabilizing protocol can be used to implement a system which tolerates transient failures.

This study shows the powerful of this formal model to deal with self-stabilization. The method allows us to transform an algorithm into a self-stabilizing one with a minimum changes since the added correction rules are able to detect and to correct transient failures. We have investigated many examples including spanning tree, SSP's algorithm, enumeration, token circulation on the ring [Dij74]. As we shall see in Chapter 5, all the algorithms presented in this chapter can be implemented and tested on the Visidia platform.

The stabilization time of these algorithms is computed in terms of the number of steps or applied rules to reach legal configuration. To measure the "real" stabilization time, we have studied the problem of resolving conflicts. We first use our approach to design self-stabilizing algorithm for resolving conflicts. The proofs that the presented algorithm converges to satisfy its specifications in a finite time is given. Then, we propose one possible implementation of the needed synchronizations using local election randomized procedures. That is, the stabilization time is approximated. The upper bounds computed during the analysis match the size of the relabeling sequence used to prove the convergence of the algorithm.

In what follows we plane several possible extensions.

1. We are interested to optimize the stabilization time which is omitted in this work.
2. The in-depth study of snap-stabilization [BDPV99, CDPV02] using graph relabeling systems. We hope that our framework allows to build elegant snap-stabilizing algorithms.
3. How to implement self-stabilizing algorithms with constraints. For example, algorithms which doesn't violate the safety properties during the stabilization phase.
4. All algorithms encoded by local computations verifying the termination detection property can be transformed into a self-stabilizing algorithms. What happen in the message passing model. The border between local computations model and the message passing model in the context of stabilization.

5. How to transform a local stabilizing algorithm to (global) self-stabilizing algorithm.
6. Assumptions about the detection mechanism, reliable or unreliable [APSV91, AKY97].
7. Code stabilization [FG05].
8. Stabilization using mobile agents.

Chapter 3

Local Failures Detection

UNreliable failure detectors were formalized for the first time by *Chandra* and *Toueg* in their seminal paper [CT96] with the consensus problem [FLP85] as motivating example. A failure detector is a local module attached to every process. Its role is to give information about the status of all other processes in the system. Typically, a failure detector maintains a list of processes that it suspects to have crashed. A failure detector can make mistakes by incorrectly suspecting a correct process, nevertheless, whenever a failure detector discovers that some process was incorrectly suspected, it can change its mind. Consequently, they are named *unreliable failure detectors*.

As locality is an important feature of distributed computing, it is essential to understand how to carry on computations based on local knowledge in the presence of crash failures. Works presented by [CS96, AKY97] introduce the notion of *failure locality* which they combined with the *number of faults* to measure the *degree of fault-detection*. Therefore, fault-detection becomes a monitoring task. We will focus in this chapter rather on the implementation of the failure detectors and investigate it by means of local computations. We exhibit a set of local computation rules to describe failure detectors. The resulted algorithm is then implemented and analyzed in the partially synchronous system. Thereby, under such assumptions our protocol is intended to implement an eventually perfect failure detector $\Diamond P$ [CT96]. Then, the protocol is extended to detect when a system component at distance l has failed and notifies the application of this state.

The method proposed to detect failures uses classical techniques to determine locally the set of faulty processes by constructing lists of suspected processes. Such a set, called the syndrome, is designed through a two-phase procedure: The test phase where processes are tested locally by their neighbors, followed by the diffusion phase where test results are exchanged among fault-free processes. Each process tries to gather information about the status of all other processes by trusting non-faulty processes. Moreover, we have developed an interface based on the Visidia library to simulate faults through a graphical user interface and visualize the dynamic execution of the failure detector algorithm. This can be implemented by equipping each process by a local failure detector module that monitors adjacent processors and maintains a set of those that it currently suspects to have crashed. Of course, the failure detector module can make mistakes by not suspecting a crashed process or by erroneously adding processes to its set of suspects. If it later believes that suspecting a given process was a mistake, it can update its set of suspects. Thus, each module may repeatedly add and remove

processes from its set of suspects.

The rest of the chapter is organized as follows. In Section 3.1, we recall some properties of failure detectors. In Section 3.2, we briefly review the existing implementations of failure detectors. The system model and assumptions used here are explained in Section 3.3. In Section 3.4, we describe our solution to implement failure detection based on local computations. An implementation and an analysis of this solution in the message passing system is proposed in Section 3.5. In Section 3.6, we present experimental results to evaluate the performance of our failure detector using Visidia platform. Then, in Section 3.7 we extend the protocol of Section 3.4 to deal with ball of radius l. Finally, Section 3.8 concludes the chapter with some open issues.

3.1 Unreliable Failure Detectors

Chandra and Toueg [CT96] gave the first formalization and classification of the unreliable failure detectors. A failure detector is a local module attached to every process. Its role is to give information about the status of all other processes in the system. Typically, a failure detector maintains a list of processes that it suspects to have crashed. A failure detector can make mistakes by incorrectly suspecting a correct process, nevertheless, whenever a failure detector discovers that some process was incorrectly suspected, it can change its mind. The authors have defined some classes of failure detectors that are very useful for solving fundamental problems in fault-tolerant distributed computing. The classification is based on specifying the *completeness* and *accuracy* properties that failure detectors in this class must satisfy. Completeness restricts the kinds of false positives, while accuracy restricts the kinds of false negatives. Each of these properties is either *weak* or *strong*; Moreover, accuracy is either *perpetual* or *eventual*. Altogether, there are eight distinct classes of failure detectors. In this chapter, we present an algorithm to implement an eventually perfect failure detector $\Diamond P$ which is characterized by strong completeness and eventually strong accuracy:

- **Strong completeness:** there is a time after which every process that crashes is permanently suspected by every correct process.

- **Eventually strong accuracy:** there is a time after which correct processes are not suspected by any correct process.

The most popular strategies for implementing failure detectors are Heartbeat and interrogation.

- **The heartbeat strategy**: each process v_j periodically sends 'I am alive' messages to the processes in charge of detecting its failure. This implementation is defined by two parameters: The heartbeat period δ and the time-out delay called *To* (see Figure 9).

- **The interrogation strategy**: a process v_i handles a process v_j by sending periodically 'Are you alive?' messages to v_j. When a process v_j receives such a message, it replies 'Yes'. This implementation is also defined by two parameters: The interrogation period δ and the time-out delay *To* (see Figure 10).

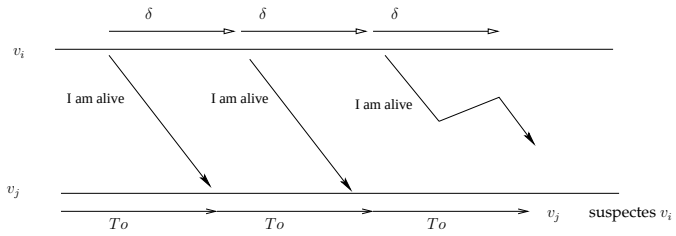


Figure 9: Heartbeat strategy

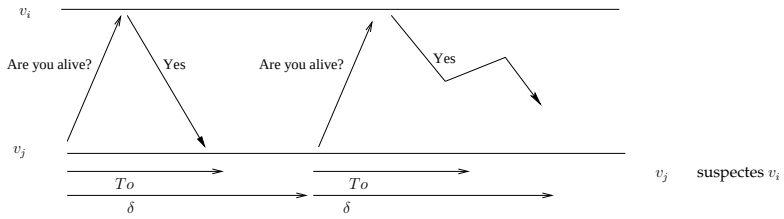


Figure 10: Interrogation strategy

3.2 Related Works

Failure detectors have been introduced in the seminal paper of *Chandra* and *Toueg* [CT96] as an approved tool to solve the consensus problem [FLP85]. In this work, they discussed the possible solutions of the consensus problem and defined the weaker family of failure detectors required to achieve this. Note, however, that the use of failure detectors don't break the FLP impossibility. That is, the implementation of the failure detector of this family requires some synchrony from the system while the algorithm using their minds may be totally asynchronous. Since then, many protocols have been already designed to implement failure detectors in different models of systems. Here we surveys some of these works.

[DFGO99] describes an interesting implementation of failure detection service which is a composition that mixes *push* and *pull* failure monitoring. Such a monitoring is similar to the heartbeat and the interrogation strategies. Further, [MMR02] proposed an implementation based on a *query/response* mechanism, and assume that the exchanged query/response messages obey a pattern where the processes-query responses arrive among the *(n-f)* first exchanged query/response messages, where n denotes the total number of processes and f denotes the maximum number of them who can crash, with $1 \leq f < n$.

Thereafter, some attention will focus on efficiency and adaptability. Thus, [FRT01] presented a protocol which is different from previous protocols: The number of failure detection exchanged messages is reduced. In fact, it uses control messages only when no application messages is sent by the monitoring process to the observed process. [BMS02] proposed an implementation of an eventually perfect failure detector $\Diamond P$, which is a variant of heartbeat failure detector adaptable to the application scalability. The implementation is based on a basic estimation of expected arrival of ' I am alive' messages and an heuristic to adapt the sending period of this messages. Similarly, many implementations of failure detection services have been proposed for specific networks. [CHK02] identifies some of the problems raised in the context of implementing failure detectors in the Grid System.

In an alternative view, many works investigate randomized failure detector implementations. The protocol proposed in [CGG01] uses two parameters: A protocol period T(in time units) and an integer k, which is the size of failure detection subgroup. The protocol steps are initiated by a member M_i, during one protocol period T. Member M_i selects at random a member M_j in the same subgroup and sends to it a *ping* message. If M_i does not receive a reply within the time-out delay, it selects k members at random and sends to each of them a *ping-req* message. Each of the non-faulty members among these k, which receives the ping-req message subsequently, pings M_j and forwards to M_i the ack received from M_j, if any. Other implementation [ACT02] includes a comprehensive set of parameters to define some metrics that characterize the quality of service of a failure detector and then its performances. In this protocol, process u sends heartbeat messages $m_1, m_2, \cdots$ to v periodically every θ time like a simple heartbeat mechanism. To determine whether to suspect u, v uses a sequence $t_1, t_2, \cdots$ of freshness points, obtained by shifting the sending time of the heartbeat messages by fixed parameter δ. Then, $t_i = e_i + \delta$, where e_i is the time when m_i is sent. For any time t, let i be such that $t \in [t_i, t_{i+1}]$, then v trusts u at time t if and only if v received heartbeat m_i or higher. For the parameters, we just give a survey of these metrics since there are not applied for our system. The model of the system used there assumes that messages get lost on a link with a given message lost probability and the message delay is given by a random variable *T* with finite expected value *E(T)* and variance *var (T)*. They define three primary metrics: *detection time*, *mistake recurrence time* and *mistake duration*.

Another issue is to consider failure detection in the context of self-stabilization. The first self-stabilization failure detector implementations were introduced in [BKM97]. Their implementations send messages with every clock tick. These algorithms satisfy the failure detector semantics and stabilize within finite time. In a recent initiative [HW05b, HW05a], a possible implementation of unreliable failure detector using self-stabilization for *message-driven* systems is proposed. A protocol is said to be message-driven if a process executes its actions (sends messages, modifies its state) only at the reception of a message. From this point of view, the implementations of [BKM97] are time-driven and not message-driven. Note, however, that the works of [HW05b, HW05a] are for partially synchronous systems.

The measures introduced in [CT96] and the most relevant protocols are global. In [HW05b], the authors formalized some characterization of the local failure detectors and gave definitions of local completeness and local accuracy. The failure detector which satisfies these two properties is self-stabilizing depending on the number of processes that may fail permanently and the timing model used. The measures are local and reduced to the neighbors. These local failure detectors are devoted for sparse networks.

Unfortunately, most of the presented works are for complete graphs, use processes identities and uses some synchrony requirements according to the desired goals. Further, they are adapted for some applications, but not for other. Which contradicts the first advantages of the use of failure detectors as detection mechanism for applications that are not concerned how the failure detector is implemented.

3.3 The System Model

Our system comprises a set of identified processes denoted $v_1, v_2 \cdots v_n$. The network is modeled by a graph $G = (V, E)$, where V is the set of processes. An edge (v_i, v_j) in E stands

for an unidirectional link from v_i to v_j. Processes communicate and synchronize by sending and receiving messages through the links. A process can fail by crashing, i.e. by permanently halting. We assume that at most $\eta \leq \#V$ processes can crash. Communication links are assumed to be reliable. As stated in the previous section, there is no way but to use a time based model to implement distributed failure detectors. In our case we use the following. For the assumption about the relative speed of processes or message transfer delay, we consider the partially synchronous model used in [CT96, BMS02, HW05b, HW05a]. Here, we abstain from a formal definition of this timing model. In general, in this model, it is supposed that, for every execution, there are bounds on process speeds and on message transmission delays. However these parameters are not known and they hold only after some unknown time GST (Global Stabilization Time). We denote Ω_{msg} the maximum message transmission delay of all messages sent after GST.

A *path* between u and v is denoted $P(u,v) = (v_0 = u, v_1, \cdots, v_l = v)$ such that $\forall i \in [0, l-1]$, $v_i \in V$ and $(v_i, v_{i+1}) \in E$. The length of a path is its number of edges, we denote this length by $Lp(u,v)$ and then $LP(G)$ stands for the longest path of G. The set of distinct paths of length less or equal to l between u and v is denoted by $DP_l(u,v)$. We use $MDP_l(u)$ to denote the minimum number of distinct paths of distance less or equal to l between u and each node of the set $NG_l(u)$. Formally, $MDP_l(u) = Min\{\#DP_l(u,v_i),$ such that $v_i \in NG_l(u)\}$.

Since the graph changes with time, we use the following assumptions. The graph required to remain connected over the whole execution. That is, we allow at most $(k-1)$ processes failing at the same time in the $k - connected$ graphs. Consequently, (locally) this means that $\forall\, u\ MDP_l(u) \geq \eta + 1$.

To encode distributed algorithms, we use both relabeling systems model and message passing model. The first one is used as a high level to describe failure detection mechanisms using relabeling rules. Here, we consider relabeling on nodes in a ball of radius 1. That is, a center of a ball may change its label according to rules depending only on its own label and the label of its neighbors. The proposed algorithms are then implemented and analyzed in the second model as described in Section 1.3.

3.4 Failure Detector Protocol Based on Local Computations

Our goal is to describe and implement the failure detector only by performing local computations, that is between adjacent processes. We assume that each process has an updated list of the states of its neighbors. More precisely, for each process v_i, we let h_i be the following set : $h_i = \{(c_j, s_j)\}$, where v_j is a neighbor of v_i, c_j is its time-stamp and s_j a Boolean value standing for its suspicious state about v_j. The time-stamp can be viewed as a local clock of process v_j, which is a counter, associated with v_j and with its state. The state s_j is set to $true$ if v_j is suspected and to $false$ otherwise. The set h_i can be also considered as a label of vertex v_i representing the process v_i. Note that h_i is a subset of $\#V \times \{0,1\}$. The Boolean value 1 means that the process is suspected of being crashed and 0 means that it is not. For simplicity, we will use the notations $(c,s)_j$ or $h_i(j)$ instead of (c_j, s_j).

Now, we define the lists of the states of all processes maintained by every processes as the following local vector labeling. To each process v_i, we associate the vector $\Gamma(i) = ((c,s)_1, (c,s)_2, \ldots, (c,s)_n)$. In fact, this vector is the current knowledge of the process v_i about the states of all other processes. We will denote the value of the j^{th} component of

the vector $\Gamma(i)$ by $\Gamma(i)[j]$. Each vector is updated by performing the following local computation. Let v_{i_0} be a process whose neighbors are $v_{i_1}, v_{i_2}, \ldots, v_{i_l}$. We use the Max function which is defined by considering the following partial order: $(c, s) > (c', s')$ if $c > c'$ or $(c = c'$ and $s = 0$ and $s' = 1)$. The initial value of $\Gamma(i)[j]$ is $(0, 0)$ for all $i, j \in [1, n]$. Clearly, each process updates its vector by taking the most recent state of each process already known by the neighbors. The computation is performed locally in the ball of radius 1. The purpose of using both h_i and $\Gamma(i)$ is to separate the local mechanism to detect failures of neighbors (stored by the h_i 's) from the global lists broadcast through the network (stored by the $\Gamma(i)$'s). As we shall detail in the next section, the former can be implemented by heartbeat or interrogation technique. Let v_{i_0} be the vertex representing v_{i_0} and whose neighbors are $v_{i_1}, v_{i_2}, \ldots, v_{i_l}$. We use the function $f_i(t)$ to denote the set of processes that have been suspected to be crashed by the process v_{i_0} at the time t. Our failure detector protocol is defined by :

Algorithm 6 Local failure detector for local computations model

FD1 : **Detection rule**

Precondition :
- $\exists\, v_{i_j} \in B(v_{i_0}),\ v_{i_j} \in f_i(t)$

Relabeling :
- $h_i(j) := (t, 1)$
- $\Gamma(i_0)[j] := h_i(j)$

FD2 : **Renaming rule**

Precondition :
- **If** $v_{i_j} \in B(v_{i_0})$

Relabeling :
- $\Gamma(i_0)[j] := Max(h_{i_0}(j),\ \Gamma(i_k)[j]),\ 0 \leq k \leq l$

FD3 : **Diffusion rule**

Precondition :
- **If** $v_{i_j} \notin B(v_{i_0})$

Relabeling :
- $\Gamma(i_0)[j] := Max(\Gamma(i_k)[j]),\ 0 \leq k \leq l$

Each failure detector executes the three rules. The rule FD1 detects suspected neighbors. We use a function f which will be implemented in the next section by a heartbeat strategy. The rule FD2 updates the local knowledge of the detector about its neighbors. Finally, the rule FD3 updates the knowledge of the detector about other processes.

3.5 Implementation and Analysis in a Message Passing System

We will give in this section an implementation of the failure detector in a distributed system where processes communicate only by exchanging messages with neighbors as explained in Section 3.3. To do so, we will translate the rules given in the previous section into algorithm using the model presented in Section 1.3. Then we will study its properties.

3.5.1 Failure Detector Algorithm

We present in this section an implementation of our protocol where $f(t)$ is accomplished by the heartbeat mechanism. The algorithm maintains the list of suspected processes as described previously. Every processes v_i periodically broadcast a message 'I am alive' to all its neighbors. If a process v_i reaches a time-out on some process neighbor v_j, it adds v_j to its list of suspected processes. To achieve these tasks, two parameters characterize this failure detector: The heartbeat period (interval) HBI and the time-out delay $HBWD$ instead of *heartbeat waiting delay*. The implementation of the algorithm is based on three lists. The list *suspected* maintains the state of the processes, it is composed of couples *(count,susp): susp* is a Boolean which is set to $true$ if the process is suspected and *count* contains the number of the last heartbeat received. The lists *lastHbeat and msgR* respectively contain the time (of the local clock) of reception of the last message heartbeat and its sequence number. Each process v_i is equipped with a local clock in order to time-stamp incoming (and outgoing) messages. Two types of messages are exchanged in the network, the heartbeat messages and lists of suspected processes messages.

The "$Suspected$" list is updated either by the mechanism of heartbeat, or after reception of a list from a neighbor. In this case the update is carried out in the following way: the state of a process is determined by the greatest number of the heartbeat as explained above.

Each process v_i executes the following algorithm, refereed to in the following as $\mathcal{FD}$, using its local clock. In the pseudo-code, the local state variables of process v_i will not necessary sub-scripted with i; But used in some proofs and discussion to avoid ambiguity.

Algorithm 7 is a direct implementation of the local computations one presented in Section 3.4. It is composed of mainly three phases. The first one is the initialization ($\mathbf{F}1$). The second one manages the heartbeat implementation ($\mathbf{F}2$ and $\mathbf{F}3$). The last one deals with the broadcast of the lists of suspected processes ($\mathbf{F}4$).

3.5.2 Properties of the Detector

We show in this section that $\mathcal{FD}$ algorithm implements a failure detector of class $\Diamond P$ in a system model as defined in Section 1.3. We consider a partially synchronous system S as explained in [CT96]. For every run of S there is a GST after which there are bounds on process speeds and on message transmission time. However, the GST and this bound are not known.

A failure detector of class $\Diamond P$ must verify the two properties described by Lemma 3.5.1 and Lemma 3.5.2 stated as follows. Let $crashed$ be the set of crashed processes and let $correct(t)$ be the set of correct processes at time t.

Lemma 3.5.1 (Strong Completeness) *The failure detection algorithm described above implements a failure detector which satisfies the following property: Eventually, each process that crashes is permanently suspected by all correct processes,* $\exists\, t_c$ *such that* $\forall t \geq t_c$, $\forall\, v_i \in correct(t)$, $\forall v_j \in crashed$, $Suspected_i[j] = < seq, true >$.

Proof. Let v_i and v_j be two neighbors. Assume that v_j crashes at time t_{crash}. It stops sending messages <**I am alive**> to its neighbors. Since we suppose a bound Ω_{msg} on the delay of message transmission, v_i will no longer receive any message from v_j after $t_{crash} + \Omega_{msg}$. The list $lastHbeat_i[j]$ contains the moment of the last message from v_j received by v_i and $msgR_i[j]$ this sequence number. Let t_{ci} be the time after which v_i suspects permanently v_j.

Algorithm 7 Local failure detector algorithm in the message passing model

var FBI /*Heartbeat interval*/;
$HBWD$ /*Time Out, Heartbeat waiting delay*/;
$Suspected$ /*List of processes which are suspected by v_i;*/

F1 : {Initialization(T1)}
$msgS = 0$ /*msgS is an integer*/;
$Suspected[j] = \langle 0, false \rangle$ for all $j = 1, \cdots \#V$;
$lastHbeat[j] = -1$ for all $v_j \in NG(v_i)$; /* $NG(v_i)$ is the set of neighbors of v_i */
$msgR[j] = 0$ for all $v_j \in NG(v_i)$;

F2 : {Heartbeat message sending:}
at time $t = HBI * msgS_i$ **do**
send<**I am alive**> to all $v_j \in NG(v_i)$;
$msgS := msgS + 1$;

F3 : {A message <**I am alive**> has arrived at v_i from v_j}
$tmax := lastHbeat[j] + HBWD$ /*The maximum time of waiting*/;
/* now is the local time of process v_i corresponding to the reception of the message*/
1: **if** $(now \leq tmax)$
$lastHbeat[j] := now$;
$msgR[j] := msgR_i[j] + 1$;
$Suspected[j] := < msgR[j], false >$;
2: **else**
$Suspected[j] := < msgR[j] + 1, true >$; /* time out*/
3: send<**Suspected**> to all $v_j \in NG(v_i)$;

F4 : {A message <**Suspected**> has arrived at v_i from v_j}
$Suspected[k] :=$ Max$((c, s), (c, s)_j)$ For all k in V;
send<**Suspected**> to all $v_j \in NG(v_i)$;

We will study two cases:

- *Case1:* If $t_{crash} + \Omega_{msg} < lastHbeat_i[j] + HBWD$ then $t_{ci} = t_{crash} + \Omega_{msg} + HBWD$.
- *Case2:* Otherwise, $t_{ci} = t_{crash} + \Omega_{msg}$.

From these two cases, it suffices to take $t_{ci} = t_{crash} + \Omega_{msg} + HBWD$. The other correct processes will detect it after the diffusion ($F3(1)$ and $F3(2)$). The time required for every correct process to suspect permanently v_j is bounded by $t_{crash}+HBWD+(LP(G)+1)\times\Omega_{msg}$. In fact, the traversal of a message in the graph requires at most $LP(G)$ steps, where $LP(G)$ is the longest path in G. Let us mention that the graph remains connected since it is k-connected and at most $(k-1)$ processes are supposed to crash at the same time. Therefore, the longest path is still defined and is finite. The value of t_c can be set to $t_{crash} + HBWD + (LP(G) + 1) \times \Omega_{msg}$. □

Lemma 3.5.2 (Eventually strong accuracy) *The failure detection algorithm described above implements a failure detector which satisfies the following property: There is a time after which no correct process is suspected,* $\exists t_a$ *such that* $\forall t \geq t_a, \forall v_i, v_j \in correct(t), Suspected_i[j] =< seq, false >$.

Proof. Suppose that v_j is a correct process and that it is suspected by process v_i, i.e. v_i does not receive the k^{th} heartbeat message of v_j. In such a case, v_i marks the j^{th} component of the list $Suspected_i$ as $true$, that is, $Suspected_i[j] =< msgR_i[j] + 1, true >$. There are two cases, depending on whether some neighbor of v_j has received the k^{th} heartbeat message or not.

- *Case1:* There is at least one neighbor v_l of v_j which receives the k^{th} heartbeat message sent by v_j (v_l may not be a neighbor of v_i), v_i carries out ($F3(2)$), starts by suspecting v_j and diffuses its list to all its neighbors which in their turns diffuse it to their neighbors and so on. Therefore, processes who had received the list sent by v_i and not yet that sent by v_l suspect v_j ($F4$). But, v_l executes ($F3(1)$), it diffuses its list in a similar way as v_i. At the end, all the processes even those which already received the list of v_i will update their lists ($F4$) and thus each correct process will no longer suspect the correct process v_j. We denote by t_d the time of correction or diffusion of the correct suspected list, $t_d = LP(G) \times \Omega_{msg}$. It suffices to take $t_a = GST + t_d$, where $LP(G)$ is the longest path of G.
- *Case2:* Otherwise, no neighbor receives the k^{th} heartbeat message from process v_j, all the neighbors suspect the correct process v_j and will diffuse these incorrect information. At least, a process neighbor v_l will receive such a message from v_j ($F3(1)$) with a possible delay bounded by Ω_{msg}. The time t_a is bounded by $GST + \Omega_{msg} + t_d$.

In both cases, after $t_a = GST + (LP(G) + 1) \times \Omega_{msg}$ no correct process is suspected. □

According to the previous Lemmas, the following theorem holds:

Theorem 3.5.3 *The failure detection algorithm described above implements an eventually perfect failure detector* $\diamondsuit P$.

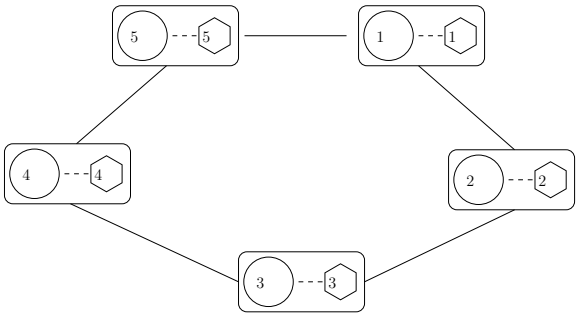


Figure 11: Network improved by failure detectors

3.6 Implementation on the Visidia Tool

To allow Visidia to detect failures, we have implemented the protocol described in Section 3.5.1. The failure detector module is implemented as a *java thread* which is attached to each node of the graph and which executes $\mathcal{FD}$ algorithm. As shown in Figure 11, the main algorithm is executed by the process computing unit, and that the detector module is executed independently. However, the suspected lists maintained by the failure detector is accessible by both.

3.6.1 Initial Implementation

This implementation allows the user to customize the tool in order to run the main algorithm alone (safe mode), the detectors alone or both at the same time. We will focus in this section rather on the implementation of the failure detector. In Chapter 5, we propose a method to simulate and to to design fault-tolerant distributed algorithms.

When the failure detectors are running, the user can observe the animation of the execution of the detectors during run-time. Dynamic traffic of messages, including $<$**I am alive** $>$ monitoring messages and lists of suspected processes, are displayed on the screen. Moreover, the user can simulate the crash of a process by selecting and marking the corresponding node by a cross. For instance, the Figure 12 shows a screen shot where the process corresponding to node 4 has been set to a crashed state. Similarly, the process of node 5 is set to a crashed state. Of course, these simulations can be done during run-time. Now, the user can visualize the current suspected lists at any node of the graph. For example, the list of suspected processes computed by node 3 shows that nodes 4 and 5 are suspected (their status are set to 1) as given in Figure 13.

3.6.2 Stability

Algorithm 7 has been implemented and initial tests show that it works correctly regardless of its expected task. The main issue now is stability, as raised in [ADGFT01], to reach better precision. The next step is therefore to test it as low level to notify applications about the status of the network components, but are not in the context of this thesis.

The goal of the tests realized here was to reach a maximum stability length. Stability is reached when the suspicion list corresponds to the real machine state. Every node holds a

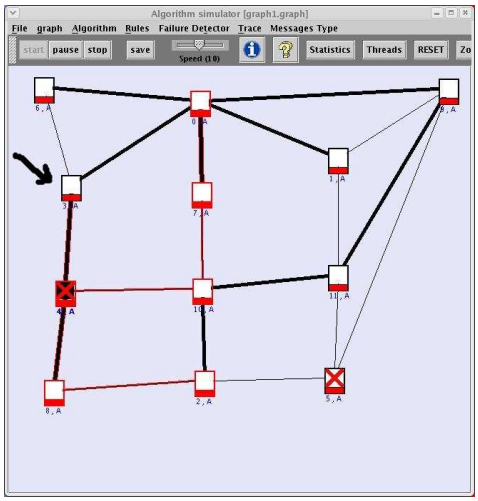


Figure 12: The simulation of crashes in Visidia

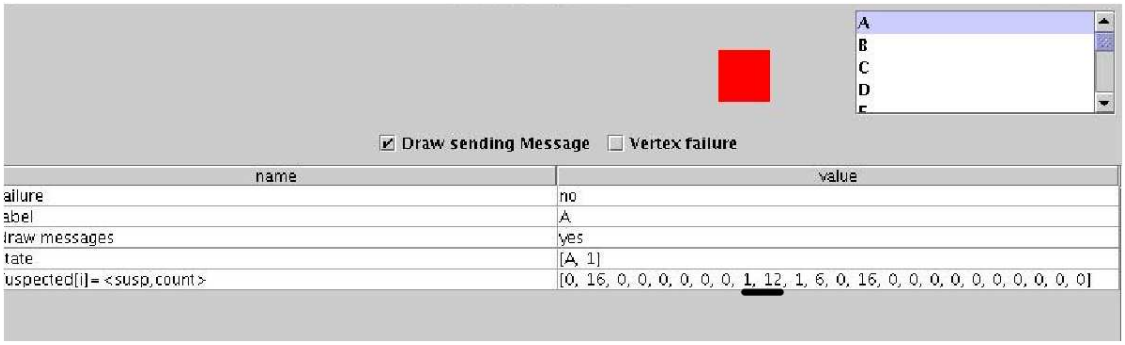
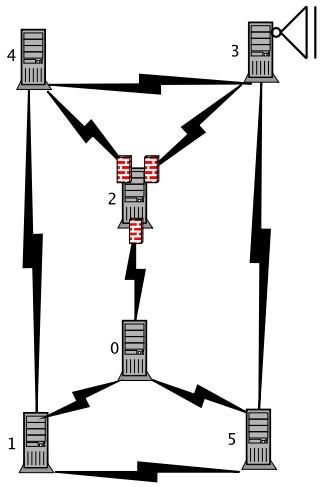


Figure 13: The suspected list computed by node 3



Experiments	Parameters	3	5	4	1	0
1	HBI	10000	8000	8000	6000	6000
	HBWD	12000	10000	10000	10000	10000
2	HBI	8000	9000	9000	8000	8000
	HBWD	14000	11000	11000	11000	12000
3	HBI	8000	5000	4000	4000	5000
	HBWD	14000	10000	9000	9000	14000
4	HBI	4000	4000	2000	2000	4000
	HBWD	12000	8000	8000	8000	12000
5	HBI	3000	3000	2000	2000	4000
	HBWD	13000	7000	8000	9000	11000
6	HBI	2000	2000	2000	2000	3000
	HBWD	12000	7000	7000	8000	11000

Figure 14: (a) Network to be tested, (b) Parameters of the experiences

suspicion list that describes the state of other nodes. As shown in the left part of Figure 14, node 2 crashed and the observations were done on the node 3 list. The right part of Figure 14 gives the parameters of the proposed experiences. At last, tests should reveal a global stability period if which all suspicion lists are stables. This is in order to define a maximum time during which processes can be ran with a total knowledge of the machine states.

The tests was to enhance Visidia with new features. Offering a homogeneous failure detection system, Visidia now offers a heterogeneous one. This heterogeneous failure detection system is based on the Heartbeat protocol and configures the detection parameters more deeply. Parameters no longer have to be the same for every machine of the network. Tests were carried on in order to approximate some stability intervals by putting the right parameters to some detector threads. Tests showed some global stability phases but mostly only local stability.

3.7 (f, l)-Failure Detection

In this section, we extend the previous protocol to deal with balls of radius more than 1. Such failure detector performs only local computations, as there are at most η processes that can crash. That is, between adjacent processes based on failure locality $f \leq \eta$ and l as the radius of balls. In the sequel, the proposed protocol will be called $\mathcal{FD}(f, l)$ algorithm. Each process has the following variables:

- $Suspected(u)$: a vector of suspected neighbors at distance l,
- $Qresp$: a set of processes that response the $query$,
- $Path$: a set of processes.

To simplify discussions, let $Push(pth, i)$ (resp. $Pull(pth)$) denotes the function to add the process i at the end of the path pth (resp. to remove the end element of the path pth) and let $high(pth)$ denotes the function to give the number of processes in the path pth.

Initially, each node is labeled (ϕ, ϕ, ϕ). Starting from an initial configuration, each node u sends message <**query**> to all its neighbors at each round. When v a neighbor of u, receives

Experiments	Results
1	There is a global stability at the end of the first 3 messages. After, two states alternates for the suspicion list: - Suspicion of all the entities. - Stability on the level of the neighbors The period of this local stability is ten messages.
2	Same results as in the first experiment but the local stability are longer and lasts 14 messages. Still no global stability.
3	After 15 messages, total stability is reached. Later, first the node 0 is suspected and then after 66 messages the node 1 is suspected. Results for this case are better than for the 1) and the 2) because there is an increase in the period of stability.
4	Results are better. The global stability is longer, the node 0 is suspected after 185 messages and the node 1 after 358. This is due to the fact that the node 5 updates its suspicion list before the others do and sends it to the node 3.
5	The node 0 is always suspected.
6	The nodes 0 and 1 are suspected very early.

Figure 15: Results of the experiences using parameters of Figure 14(b)

such a message, it responses by sending message <**ok**> and forwards the query of u to its neighbors except u if there is other processes at distance l ($high(pth) < (l-1)$). If w which is not a neighbor of u receives a query about u from v, it responses to v and v forwards the response to u. When u receives $\#NG_l(u) - \eta$ distinct responses, it starts to suspect the other η processes

3.7.1 Implementation in a Message Passing System

Here, we present an implementation of the $\mathcal{FD}(f, l)$ algorithm in the message passing system. We introduce the following variables:

- $round(u)$: gives the current round of the process u. Initialized to 0 and each process increases its variable $round$ after the reception of $\#NG(u) - \eta$ messages from its neighbors in the ball of radius l,

- $chrono$: a local clock to implement a waiting procedure.

We consider a timing model near the partially synchronous system as explained in [CT96]. For every run of a distributed computing under a system as described in Section 1.3, there is a global stabilization time GST after which there is an upper bound $\tau < \infty$ on message transmission delay. In the following, we present the failure detector protocol. Each failure detector executes four actions. Action F_l1 sends message <**query**> each round to all its neighbors and the action F_l2 receives a response <**ok**> for such a message. Action F_l3 responses message <**query**> and sends it to all other processes in the ball of radius l. Finally, the action F_l4 starts to suspect the processes which do not respond and initiates a new round. In the worst case, the <**query**> message takes $l\tau$ time from an initiator to the most distant neighbor in the ball of radius l. Therefore, it suffices to use $2l\tau$ as the time to detect the failures which is the time to send a query and to receive a response from the most distant neighbors.

Algorithm 8 (f, l) Local failure detector algorithm in the message passing model

var $round$: integer **init** 1; $chrono$: integer **init** 0;
$Susp$, $Qresp$, $path$: set of labeled edges **init** ϕ;

$\mathbf{F}_l1$: {At each round, v_0 executes:}
for (v_i **IN** $NG(v_0)$) **do**
send<**(query,**ϕ**)**> via e_i

$\mathbf{F}_l2$: {A message <**(ok,path,**e_l**)**> arrived at v_0 from e_j}
if ($e_l \notin Qresp$)
$Qresp := Qresp \cup \{e_l\}$;
if ($high(path) = 0$)
$chrono := chrono + 1$;
else / $* high(path) > 0 *$ /
send<**(ok,path,**e_l**)**> via $Pull(path)$

$\mathbf{F}_l3$: {A message <**(query,path)**> arrived at v_0 from e_j}
send<**(ok,path,**e_0**)**> via e_j
if ($high(path) < (l-1)$ and $NG(v_0) \neq e_j$)
for (v_i **IN** $NG(v_0, 1) \backslash \{v_j\}$) **do**
send<**(query,push(path,**e_j**))**> via e_i

$\mathbf{F}_l4$: {On $chrono \geq 2l\tau$ and $\#Qresp \geq \#NG_l(v_0) - \eta$ /*end of chrono*/}
$Susp := NG_l(v_0) - Qresp$;
$round := round + 1$;
$(Qresp, chrono) := (\phi, 0)$;

3.7.2 Analysis

To do the algorithm's analysis, we redefine some of the concepts introduced above, these definitions are given using the time.

Definitions 3.7.1 *The following notations are used in the sequel:*

1. $round(u, t)$ *is the value of variable* $round$ *of* u *at time* t*,*
2. $Suspected(u, t)$ *is the set of processes that are suspected by* u *at time* t*,*
3. $q(u, t)$ *denotes the last query invoked by* u *that has terminated at or before* t*,*
4. $Query\text{-}resp(u, t)$ *is the set of* $\#NG_l(u) - \eta$ *processes from which* u *has received the responses* <**ok**> *to* $q(u, t)$.

We denote by $\Diamond P_{(f,l)}$ the class of eventually perfect l-local f-failure detectors which satisfy l-local completeness and eventually l-local accuracy properties. These definitions are given in the following:

Definition 3.7.2 ***(l-Local Completeness)*** *Let v be any node in the graph. If v is correct, then every node that crashes in the ball $NG_l(v)$ is eventually suspected by v.*

Definition 3.7.3 ***(Eventually l-Local Accuracy)*** *Let v be any node in the graph. If v is correct, then every node that is correct in the ball $NG_l(v)$ is not eventually suspected by v.*

From the fact that $MDP_l(u) \geq \eta + 1, \forall\, u \in V,$ it results that the graph is $\eta + 1$-connected, meaning that $\forall\, (u, v) \in V^2,$ there is at least $\eta + 1$ *different* paths. Let v be any node in the

graph G, we denote by $\mathcal{V}al(u,l)$ the set of nodes in $NG_l(u)$ such that $\forall\, v \in \mathcal{V}al(u,l)$, there is at least a *correct* path $P = (v_0 = u, v_1, \cdots, v_l = v)$ such that $\forall\, i \in [1, l-1]$, v_i is correct and $v_i \in NG_l(u)$. Formally,

$$\begin{aligned}\mathcal{V}al(u,l) \quad = \quad & \{v \in NG_l(u) \mid \exists\, P = (v_0 = u, v_1, \cdots, v_l = v) \\ & \text{s.t. } \forall\, i \in [1, l-1],\ v_i \text{ is correct and } v_i \in NG_l(u)\}\end{aligned}$$

In the following, we will just deal with nodes of the sets $\mathcal{V}$al called valid nodes.

Lemma 3.7.4 *The $\mathcal{FD}(f,l)$ algorithm verifies l-*local completeness *property.*

Proof. Let u be a correct node, and v a node in $\mathcal{V}al(u,l)$. From the discussion above, there is at least one correct path P from u to v such that all the nodes in P are correct and belong to $NG_l(u)$. Let w be a crashed node in $NG_l(u)$. From the action F_l4 u will receive $\#NG_l(u) - \eta$ responses to its query, then we distinguish two cases. If node w crashes after the send of its response and the case of it crashes before the send of the response. For the first case, u will start to suspect in the next round. For the second case, u start to suspect w in the current round (action F_l4). □

Lemma 3.7.5 *The $\mathcal{FD}(f,l)$ algorithm verifies* eventually l-local accuracy *property.*

Proof. Let u and v be two nodes such that $v \in NG_l(u)$. From the algorithm, u will start to suspect v after $2l\tau$ time and when $\#Query-resp(v) = \#NG_l(u) - \eta$. Once each correct valid path $P = (v_0 = u, v_1, \cdots, v_l = v)$ is of length l less than l, u will never start to suspect v. □

From the two previous Lemmas, we state the following result:

Theorem 3.7.6 *The $\mathcal{FD}(f,l)$ algorithm implements an eventually perfect l-local f-failure detector denoted by $\diamondsuit P_{(f,l)}$.*

3.8 Status and Future Works

This chapter deals with the problem of crash failures and investigate it by means of local computations. We are interested here rather in the detection phase. Thus, we exhibit a set of local computations rules to describe failure detectors and then we propose an encoding of such a protocol in the message passing model. Therefore, its implementation is done in the Visidia platform.

We have presented a failure detection algorithm for local computations. The protocol designed has a two-phase procedure: A test phase where processes are tested locally by their neighbors using a heartbeat strategy (note that the interrogation can be also used), followed by a diffusion phase where test results are exchanged among fault-free processes.

The impossibility to implement reliable failure detection protocol in asynchronous distributed systems may be reduced to the impossibility of a consensus problem [FLP85, CT96]. Hence, there is no way but to use a time based model. Therefore, we relax the model using a partial synchrony model. The previous protocol is implemented and analyzed in the message passing model using such a timing assumption. It is an eventually perfect failure detector $\diamondsuit P$ [CT96]. Specifically, we prove that if the failures are permanent, it takes the longest path of the underlying graph for the failure detector to maintain the same suspected processes. Thus, in some way we can consider this distance as the time complexity of the failure detector.

The second contribution consists in the extension of this protocol to deal with ball of radius $l \geq 1$. So we extends the properties to capture such structures. The proposed protocol implements an eventually perfect l-local failure detector denoted by $\Diamond P_{(f,l)}$. This work generalizes previous work of [HW05b, HW05a] and encodes those of [CT96]. We hope that our $\Diamond P_{(f,l)}$ protocol will be used to capture faults in balls of radius l.

The failure detection service is integrated to Visidia. Thus, this tool is expected to yield an homogeneous and general environment to implement and test fault-tolerant distributed algorithms. The reliability of such algorithms depends on the used failure detector while the locality is not violated. As we shall see in Chapter 5 the presented protocols are suitable to design fault-tolerant applications. During the test phases, we proposed the use of heterogeneous failure detector instead of homogeneous one. Particularly, we investigate the notion of local and global stability of the used failure detectors. In fact, for some applications needing a reliable network during the execution of their critical transactions (routing updates, maintenance of distributed database, ...) the search of a global stability interval would be useful to produce the required quality of services.

Unreliable failure detectors are characterized by the kinds of mistakes they can make: (1) not suspecting a crashed process, (2) incorrectly suspecting a correct process. Then, unreliable failure detectors from a given class are characterized by properties about their completeness and their accuracy. Currently, we investigate another view of failure detectors measures according to the number of times they change their suspicion value about all or some processes. We need two procedures: First an algorithm to detect (or suspect) failure which allow to each process to compute its mind. Second, a mechanism to update or to consolidate such a mind. To encode the first procedure, we can use the $\mathcal{FD}$ algorithm presented above. For the second one, we propose to use an algorithm such as SSP algorithm [SSP85] already presented in the previous Chapter. We plan to implement such a protocol and then to propose a framework to reconfigure it in order to simulate the failure detectors of [CT96].

Using this new view, we hope that we can formulate questions like "what is the number of failure detectors of some classes required to solve a give problem" which are not directly expressible in conventional models to encode distributed algorithms in unreliable networks.

Chapter 4

Local Fault-tolerance

Fault-tolerance and particularly local fault-tolerance is an attractive field in distributed computing. In fact, fault-tolerance research covers a wide spectrum of applications ranging across embedded real/time systems, commercial transaction systems, transportation systems, and military/space systems, to name a few. On the other hand, as networks grow fast, dealing with fault-tolerance globally is no longer feasible. The solutions that deal with local fault-tolerance are rather essential because they are scalable and can be deployed even for large and evolving networks efficiently. A system which continues to function even after failure of some of its components is qualified as fault-tolerant system, or commonly "reliable system".

From the literature, we can distinguish between two principal approaches to improve the reliability of a system. The first is called *fault prevention* [Lap92] and the second approach is *fault-tolerance* [AG93, Sch90, AAE04]. The aim of the last approach is to provide a service in spite of the presence of faults in the system. We will be interested in this approach. In an other point of view, we agree with *G. Tel* [Tel00] about the distinction between two approaches to deal with faults in distributed computing systems: Self-stabilization and robustness. So the first one will applied to deal with transient failures and the second one, as we will see, will be applied with some refinements for crash failures. Robust algorithms provide their functionality even in the presence of faults. Note, however, that the class and the number of faults usually have to be known and limited in order to build system satisfying the required properties. As stated in Chapter 2, self-stabilization is able to recover from arbitrary system in spite of error state in finite time. Thus, self-stabilization allows to tolerate transient failures. Unfortunately, in such a system the service is only guaranteed during *stable* periods corresponding to the periods without new state corruption occurrence.

In this chapter, we study permanent failures, called also *crash failures*, and we investigates them in our framework: Local computations model [LMS99]. That is, processes may fail by crashing, i.e., they stop executing their local program permanently. We use *fault-tolerance* instead of methods and techniques to build robust algorithms. Failure detectors are usually studied and used for robust algorithms. Thus, we belief that the combining of self-stabilization and unreliable failure detectors improve fault-tolerance following three aspects: First, it makes easier the design of such protocols, second the designed protocols become not dependent from the kind and the number of failures, third the designed protocols are more robust and efficient regardless of the frequency of failure occurrence.

We deal with the problem of designing algorithms encoded by local computations on a distributed computing with crash faults. With this purpose in mind, we present a formal method based on graph rewriting systems for the specifications and the proofs of fault-tolerant distributed algorithms. In our approach, the properties of the program in the absence of faults are encoded by a rewriting system, and the fault-tolerance properties of the program are described with the behavior of the program when some faults occur. The faults are specified as a set of illegitimate configurations that disturb the state of the program after crashes of some component. We propose an operational and practical methodology to construct fault-tolerant protocols. This methodology is illustrated by an example of a distributed spanning tree construction. This example showed that the proposed method is more efficient, in the worst case equivalent, to the general method to correct the distributed system based on the reset of the entire system after a crash fault is detected [AG90].

Our purpose to deal with fault-tolerance in distributed computing using means of local computations is to propose an homogeneous method and framework to deal with such environment. Thus, as we shall see in the next Chapter the designed algorithms are implementable and testable in visidia software. The method proposed here completes the unreliability of failure detection service by the self-stabilization property of the algorithms. Since many methods are based on the initialization of the computation after each failure occurrence: Our framework uses locality because we would like to preserve as much as possible some computations that are for example far from the regions of the faults. So the initialization concerns only the balls closed to the faults.

The outline of the chapter is as follows. Section 4.1 starts with a presentation of the well-known impossibility of the consensus problem in asynchronous distributed computing systems. In Section 4.2, we briefly survey the existing solutions. In Section 4.3, we present the system model and assumptions required to our considerations. Section 4.4 describes our method to design fault-tolerant systems using local computations with illegitimate configurations, an extending model to represent the faulty processes. Then, Section 4.5 presents example of fault-tolerant spanning tree with termination detection algorithm, an application of our approach. Section 4.6 summarizes the chapter including our findings and some further researches.

4.1 Impossibility of the Distributed Consensus

In this section we present briefly one of the problem which remains unsolvable in the most model of distributed computing systems. As introduced in Section 0.2.2, the well-known result about the impossibility of a distributed consensus with one crashed process in asynchronous message passing model is given in [FLP85]. Since that, this problem has gained much attention in distributed computing since in one hand many other problems may be reduced to such a problem and in the other hand this problem is the base on many more complicated protocols as distributed data bases. The work of [AAL87] extends the same result of [FLP85] in shared memory model. Recently [VÖ4] presents a new formalization of the consensus problem without using the notion of *bivalency*: A configuration is said to be bivalent if two different final configurations may be reached from the same initial configuration. The work of [VÖ4] is given also in asynchronous message passing model. The proofs given in all of these works are based on the intuition that in asynchronous network, a crashed process cannot be reliably distinguished from one which is merely slow.

Most existing solutions for the consensus problem have included randomization, synchronizations, and failure detectors. For those using the last method, the well-known solutions are often built using the *rotating coordinator*. In [CT96], they proposed various implementations of failure detectors to solve the consensus problem. Along these propositions, we briefly review the basis of an algorithm using an eventual strong failure detector $\Diamond S$. Let n, f denotes respectively the total number of processes and the maximum number of them who can crash. The algorithm works as follows. In every *round* r processes $u = (r \ mod \ n) + 1$ is the coordinator. All processes send their estimate to the current coordinator and wait either for answer or that failure detector suspects the coordinator. The coordinator waits for $(n + 1)/2$ (the majority of correct process is needed: $f < n/2$) estimates, carefully selects one of them and broadcast this value back to all processes. In a second acknowledgment phase the coordinator detects whether agreement has been achieved or not and in case, reliably broadcasts the decision value. Other works including those of [ADGFT03, MOZ05] proceed similarly except that the coordinator does not rotate but is always the current leader (which is changed when is need, in other words when the current leader crashes). In all the coordinator based approaches, eventually every process trusts a coordinator. In this case, the agreement can be done. Note, however, that the perpetual failure detector proposed in [CT96] can be used to solve the consensus problem for $f < n$ crashes. The algorithm proceeds in asynchronous rounds, where the failure detector is used to avoid that process waits infinitely long a crashed process. The solution built there is based on the exchanges of a vector of estimates of the initial values of the processes for $f + 1$ rounds.

Note, however, that the proposed solutions are for complete communication network and processes have identities.

4.2 Related Works

As presented in [Tel00], robust algorithms are designed in order to provide their services even in the presence of crash faults. Before the introduction and the formalization of the failure detectors, the general method to correct the distributed system was the reset of the entire system after a crash fault is detected [AG90]. This approach allows to solve very general problems that appear frequently as sub-tasks in the design of distributed systems. These elementary tasks include broadcasting of information, election of a leader, mutual exclusion, spanning tree construction, etc. The basic techniques that followed are to limit the number of crashes as used in [LSP82] or to introduce some weak forms of synchrony as defined in [FLM85].

As stated previously, the use of a weak forms of synchrony has formalized and abstracted in the failure detectors oracle as presented in [CT96]. For this system, the consensus problem can be solved in a fault-tolerant manner. Such a construction may be seen as a gap between the synchronous of the system and the failure detector information.

In [AG93], a formalization of fault-tolerance is investigated through simple examples. It is based on convergence and closure properties. Still, the two basic approaches to design fault-tolerant systems can be characterized as either restricting the system (the kind of faults, their number and duration) or extending the model (timing model, synchronous, failure detection) as given in [Sch90, JL99, LM94, AAE04]. Further, most of the works proposed in the literature [AH93, Gar99, Por99, AK00] propose global solutions which require to involve the entire system.

Failure locality is introduced in [CS92, CS96] as a fault-tolerance algorithm metric which quantifies the concept as the maximum radius of impact caused by a given fault. This is an attractive method to improve fault-tolerance. In this area, we try to isolate the side-effects of faults within local neighborhoods of impact. The first consideration of failure locality was for dining algorithms [CS92]. The proposed algorithm used various *doorway* mechanisms to ensure progress and to improve the failure locality and the lower bound of 2 is established for asynchronous dining. In a recent initiative, [PS04] investigated the dining philosophers problem and proposed general and efficient solutions. They improved the system by an eventually perfect failure detector (see Section 3.5.1) to construct an algorithm with crash locality 1. Note, however, that the implementation of such failure detector requires partial synchrony. Still, this algorithm matches the lower bound of [CS92, CS96] for asynchronous system. In these works ([CS92, CS96, PS04]), the failure locality is formalized using the two classical properties related to the dinning problems: *Safety* and *progress*. Specifically, in [PS04] a pseudo-transformation is started to express these properties and then the failure locality in terms of closure and convergence as stated in [AG93].

In an other hand, in [KP95] the notion of fault local *mending* was suggested to design fault tolerant algorithms that scale to large networks. Rather, for a fault handling mechanism to scale to large networks, its cost must depend only on the number of failed nodes. Moreover, it should allow the non-faulty regions of the networks to continue their functioning even during the recovery of the faulty parts. That is, for such algorithms the complexity of recovering is proportional to the number of faults. We believe that this work is the first introduction to the concept of fault-local algorithms. Note, however, that this work is motivated with the design of algorithms whose cost depends only on the number of faults, which may be unknown. The demonstration of the feasibility of this approach is showed via developing algorithms for a number of key problems, including *MIS*. Then, in [KP00] this notion (fault local mending) is refined using the concept of *tight* fault locality to deal with problems whose complexity (in the absence of faults) is sub-linear in the size of the network. For a function whose complexity on an n-node network is $f(n)$, a tightly fault local algorithm recovers a legal global state in $O(f(x))$ time when the (unknown) number of faults is x.

To summarize this brief review, as networks grow fast, detecting and correcting errors globally is no longer feasible. The solutions that deal with local detection and correction are rather essential because they are scalable and can be deployed even for large and evolving networks. However, it is useful to have the correct (non faulty) parts of the network operating normally while recovering locally the faulty components. Consequently, it becomes intrinsically difficult to implement localizing techniques accurately. Thus, most related works in the literature suffer from poor failure locality. Additionally, it will be enjoyable to build methods which make as much as possible to keep a part of the calculations already carried, contrary to the use of methods of initialization. Moreover, few works consider the combination of failure detection service and self-stabilizing algorithms. Thus many topics are still not studied in this field.

4.3 The System Model

The distributed system (network) is a set of processes and communication links. Processes (or nodes) communicate and synchronize by sending and receiving messages through the links

(instead of edges). The graph modeling the network is unspecified and each node communicates only with its neighbors. Two nodes are considered as neighbors if and only if there is a link (channel) between the corresponding processes. We deal exclusively with connected topologies. A process can fail by crashing, i.e. by permanently halting. Communication links are assumed to be reliable. The notions of crash failure and correct process are similar to those given in the previous Chapter. Here, the system is improved by failure detector modules. Thus, after a node fails, an underlying protocol, implemented using this modules, notifies all neighbors of this node about the failure.

In our approach, the properties of the program in the absence of faults are encoded by a rewriting system, and the fault-tolerance properties of the program are described with the behavior of the program when some faults occur. The faults are specified as a set of illegitimate configurations that perturb the state of the program [AG93]. A fault-tolerant distributed system is thus the composition of two rewriting systems: The first one is the system which codes the application in a reliable system (without faults). The second one is that making it possible to code some correction rules. Here we use relabeling on nodes in a ball of radius 1. That is, each node in this ball may change its label according to rules depending only on its own label and the labels of its neighbors.

In this chapter, processes may fail by crashing, i.e., they stop executing their local program permanently. In order to capture this fact, we introduce the following:

Definitions 4.3.1

- *The crash failure of some process v_i in the local computations model is modeled using a rule $crash_i$. After the execution of such a rule, process v_i stops all its tasks(it does not execute any other rules) and prevents other processes to evaluate its states. These knowledge are used only to define failure in this model, but don't affect any other processes and the resulting state of process v_i is indistinguishable.*

- *A process v_i is said to be correct if it does not execute $crash_i$ rule.*

As before, processes cannot crash at will, but only as the result of a crash fault action. Note, however, that the network is partially synchronous regardless of the implementation of the failure detection service and asynchronous for the algorithm. Furthermore, the network is identified for the first implementation and is semi-anonymous for the studied algorithms: Only the root of the spanning tree needs to be identified.

Here, we motivate our study on the construction of a spanning tree of a graph modeling an unreliable network. Therefore, we assume the existence of a distinguished node (the root) which is usually not crashed. The graph required is assumed to remain connected during the whole execution, we allow at most $(k-1)$ failing processes at the same time in the $k-connected$ graphs. Such assumption is not a loss of generality, since it is not useful to construct a spanning tree for the non connected graph. Examples given in this Chapter are such requiring connectedness of the graph: We are interested to study the fault-tolerant distributed algorithms where the connection of the graph guarantees the existence of solution. Then, the parameter $(k-1)$ is in some way the only required fault-tolerance degree [AH93] of these algorithms.

4.4 Local Fault-tolerance

After the crashes of some components in the distributed system, some other components become temporarily faulty: Incoherence is introduced in the execution of the algorithm because of both the failure of some components and the mistakes caused by their corresponding failure detectors.

Definitions 4.4.1 *We use the following definitions:*

- *Crashed process: a process permanently stops its execution after a crash. It does not follow its algorithm.*
- *Faulty process: a process which is contaminated by a crashed process. It follows its algorithm but may deviate from that prescribed by its algorithm.*
- *Correct process: a process which does not belong to the set of crashed processes nor to the set of faulty processes. It follows its algorithm.*

From previous definitions, *fault-tolerance* is the mechanism to recover the faults (errors) introduced after the crash of some components during the computation in the distributed systems. The contaminated processes are the processes which do not respect the specification of the system after the crashes occur. These processes are the neighbors of crashed processes and we are interested to eliminate locally these bad (illegitimate) configurations.

4.4.1 A Model to Encode Fault-tolerant Distributed Algorithms

A *local configuration* of a process is composed of its state, the states of its neighbors and the states of its corresponding edges. In this work, we will be interested in local illegitimate configurations. Recall from Chapter 3 that a graph relabeling system with illegitimate configuration (GRSIC) is a quadruple $\mathcal{R} = (\Sigma, \mathcal{I}, \mathcal{P}, \mathcal{F}),$ where Σ is a set of labels, $\mathcal{I}$ is a subset of Σ called the set of initial labels, $\mathcal{P}$ is a finite set of relabeling rules and $\mathcal{F}$ is a set of illegitimate configurations.

Let us give an examples of illegitimate configuration. Consider the following graph relabeling system given to encode a distributed spanning tree. Every node v_i is labeled $L(v_i) = (span(v_i), par(v_i))$ and then it maintains the two following variables:

- $span(v_i)$: is a variable which can have two values:
  - A: v_i is in the tree
  - N: v_i is not yet in the tree
- $par(v_i)$: is the port number of the parent of v_i in the spanning tree, i.e the node which causes the inclusion of v_i in the tree.

An elementary step in this computation may be depicted as a *relabeling step* by means of the relabeling rule RST1, given in the following, which describes the corresponding label modifications (remember that labels describe process status):

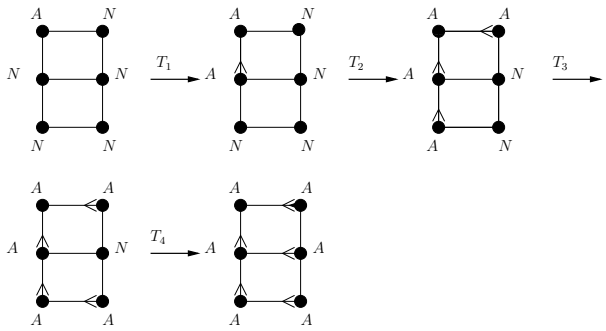


Figure 16: Example of a distributed spanning tree's computation without faults

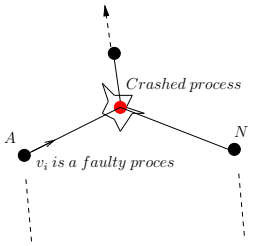


Figure 17: The faulty process with the crashed process

RST1 : **Spanning rule**

Precondition :

- $span(v_0) = N$
- $\exists\, v_i \in B(v_0), span(v_i) = A$

Relabeling :

- $span(v_0) := A$
- $par(v_0) := v_i$

Initially, all the nodes are labeled $(N, \perp)$ except one chosen node (the root) labeled $(A, \perp)$. Whenever a N-labeled node finds one of its neighbors labeled A, then the corresponding subgraph may rewrite itself according to the rule. After the application of the relabeling rule, node v_0 labeled $(N, \perp)$ changes its label to (A, v_i) where v_i is its neighbor labeled A. A sample computation using this rule is given in Figure 16. In this figure, the value of the variable $span(u)$ is the label associated with the node. The value of the variable $par(u)$ is shown by $\uparrow$. Relabeling steps *may* occur concurrently on disjoint parts on the graph. The set E_m is the set of edges $(v_i, par(v_i))\ \forall\ v_i \in V$. When the graph is irreducible, i.e no rule can be applied, a spanning tree of a graph $G = (V, E)$ is computed. This tree is the graph $G_t = (V, E_m)$ consisting of the nodes of G and the set of the marked edges.

The previous algorithm can be encoded by the relabeling system $\mathcal{R}_1 = (\Sigma_1, \mathcal{I}_1, \mathcal{P}_1)$ defined by $\Sigma_1 = \{\{N, A\} \times \{1 \cdots deg(G) \cup \{\perp\}\}, \mathcal{I}_1 = \{\{N\} \times \{\perp\}\}$ and $\mathcal{P}_1 = \{RST1\}$.

Clearly, a node labeled A must have a parent, if $span(v_i) = A$ and v_i is not root, then there exists at least one neighbor of v_i labeled A, or a parent of v_i is crashed and v_i is a faulty process as shown in Figure 17. Formally, we deal with the following predicate $f_1 : span(v) = A$, $v \neq$ root $and\ \neg\exists\, u \in B(v) : span(u) = A$.

4.4.2 Local Fault-tolerant Graph Relabeling Systems (LFTGRS)

A local fault-tolerant graph relabeling system is a triple $\mathcal{R} = (\Sigma, \mathcal{P}, \mathcal{F})$ where Σ is a set of labels, $\mathcal{P}$ a finite set of relabeling rules and $\mathcal{F}$ is a set of illegitimate local configurations. A local fault-tolerant graph relabeling system must satisfy the two following properties:

- *Closure:* $\forall (G, L) \in \mathcal{G}_L$, if $(G, L)\neg \vdash \mathcal{F}$ then $\forall (G, L')$, such that $(G, L) \xrightarrow[\mathcal{R},*]{} (G, L')$, then $(G, L')\neg \vdash \mathcal{F}$.
- *Convergence:* $\forall (G, L) \in \mathcal{G}_L$, $\exists$ an integer l such that $(G, L) \xrightarrow[\mathcal{R},l]{} (G, L')$ and $(G, L')\neg \vdash \mathcal{F}$.

As for fault-tolerant algorithms, the closure property stipulates the correctness of the relabeling system. A computation beginning in a correct state remains correct until the terminal state. The convergence however provides the ability of the relabeling system to recover automatically within a finite time (finite sequence of relabeling steps). As we shall see, the set of relabeling rules $\mathcal{P}$ is composed by the set of relabeling rules P used for the computation and some correction rules P_c that are introduced in order to eliminate the illegitimate configurations. The latter rules have higher priority than the former in order to eliminate faults before continuing computation.

As we shall see, the set of relabeling rules $\mathcal{P}$ is composed by the set of relabeling rules P used for the computation and some correction rules P_c that are introduced in order to eliminate the illegitimate configurations. The latter rules have higher priority than the former in order to eliminate faults before continuing computation.

Theorem 4.4.2 *If $\mathcal{R} = (\Sigma, \mathcal{I}, \mathcal{P}, \mathcal{F})$ is a graph relabeling system with illegitimate configurations (GRSIC) then it can be transformed into an equivalent local fault-tolerant graph relabeling system (LFTGRS) $\mathcal{R}_s = (\Sigma, \mathcal{P}_s, \mathcal{F})$.*

Proof. We will show how to construct $\mathcal{R}_f = (\Sigma_f, \mathcal{P}_s, \mathcal{F}_f)$. It is a relabeling system with priorities. To each illegitimate local configuration $(B_f, L_f) \in \mathcal{F}_f$, we add to the set of relabeling rules the rule $R_c = (B_f, L_f, L_i)$ where L_i is a relabeling function associating an initial label with each node and edge of B_f. The last relabeling function depends on the application; for example, the initial value of a node label is N in general, and the label of an edge is 0. The rule R_c is, in fact, a correction rule. Thus the set of $\mathcal{P}_f$ consists of the set $\mathcal{P}$ to which is added the set of all correction rules (one rule for each illegitimate configuration). Finally, we give a higher priority to the correction rules than those of $\mathcal{P}$, in order to correct the configurations before applying the rules of the main algorithm. It remains to prove that it is a fault-tolerant system.

1. *Closure:* Let $(G, L)\neg \vdash \mathcal{F}_f$. If (G, L') is an irreducible graph obtained from (G, L) by applying only the rule of $\mathcal{P}$, then (G, L') does not contain an illegitimate configuration. This can be shown by induction on the sequences of relabeling steps [MMS02, LMS99].
2. *Convergence:* Let $\mathcal{G}_{\mathcal{L}}$ be the set of graphs G and $h : \mathcal{G}_L \longrightarrow \mathbb{N}$ be an application associating with each graph G, the number of its illegitimate configurations, then for a graph (G, L), we have the following properties:
   - The application of a correction rule decreases $h(G)$.
   - The application of a rule in P does not increase $h(G)$.

Since, the correction rules have higher priority than the rules in P, and since the function h is decreasing, then it will reach 0 after a finite number of relabeling steps.

Note that the last property of convergence can also be proved by using the fact that the relabeling system induced by the correction rules is noetherian. Let us note that the correction rules depend on the application. While the proofs above are based on the local reset (to the initial state) which can be heavy because it may induce a global reset by erasing all the computations, it is more efficient for particular applications to choose suitable corrections as we shall see in the following examples.

We present in the sequel a spanning tree computed by a local fault-tolerant graph relabeling system. In order to define the set of illegitimate configurations to construct a set $\mathcal{F}_f$, we propose the following definition:

Definition 4.4.3 (correct node (faulty)) *A node v is correct (resp. faulty) if it satisfies one (resp. it satisfies none) of the following properties:*

1. *if v is labeled $(A, \perp)$ then $v = root$,*
2. *if v is labeled (A, u) then there exists one node u labeled (A, w),*
3. *if v is labeled $(N, \perp)$ then there does not exist node u labeled (A, v).*

Thus, for the present system we deal with the following set: $\mathcal{F}_{f_1} = \{f_1, f_2\}$, where f_1 and f_2 are defined as:

1. $f_1 : \exists\, v_0 \neq$ root, $span(v_0) = A$ and $\neg\exists\, v_i \in B(v_0) : par(v_0) = v_i \; and \; span(v_i) = A$.
2. $f_2 : \exists\, v_0$, $span(v_0) = A$, $par(v_0) = v_i$ and $span(v_i) = N$.

The correction rules are deduced from the previous configurations. Therefore, we obtain the following algorithm:

Algorithm 9 Local fault-tolerant computation of a spanning tree

RST1 : **Spanning rule**

Precondition :
- $span(v_0) = N$
- $\exists\, v_i \in B(v_0), span(v_i) = A$

Relabeling :
- $span(v_0) := A$
- $par(v_0) := v_i$

RF_ST1 : **Crash of a parent rule**

Precondition :
- $v_0 \neq$ root
- $span(v_0) = A$
- $\neg\exists\, v_i \in B(v_0) : par(v_0) = v_i \; and \; span(v_i) = A$

Relabeling :
- $span(v_0) := N$
- $par(v_0) := \perp$

RF_ST2 : **Cleaning rule**

Precondition :
- $span(v_0) = A$
- $par(v_0) = v_i$
- $span(v_i) = N$

Relabeling :
- $span(v_0) := N$
- $par(v_0) := \perp$

We assume in this system the existence of a distinguished node called the root which is initially labeled A and which is usually correct. The graph to which the algorithm is applied is k-connected and at most $(k-1)$ failures may occur. We define the relabeling system $\mathcal{R}_{f_1} = (\Sigma_{f_1}, \mathcal{P}_{f_1}, \mathcal{F}_{f_1})$, where $\mathcal{P}_{f_1} = \{RST1, RF\_ST1, RF\_ST2\}$ such that $RF\_ST1, RF\_ST2 \succ RST1$. We now state the main result.

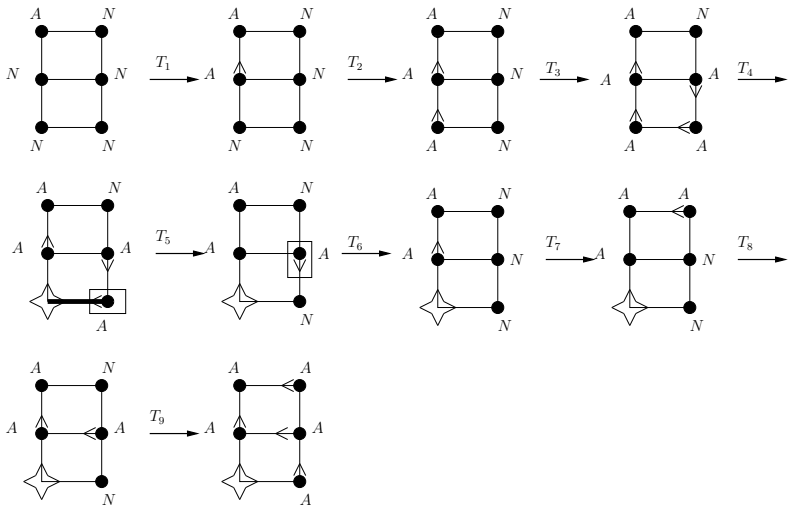


Figure 18: Example of fault-tolerant spanning tree algorithm execution

Theorem 4.4.4 *The relabeling system $\mathcal{R}_{f_1}$ is locally fault-tolerant. It encodes a fault-tolerant distributed algorithm to compute a spanning tree.*

Proof. The proof of fault-tolerance results from Theorem 4.4.2. To show that the result is a spanning tree, we use the following invariants which can be proved by induction on the size of the relabeling sequences:

(I1) All N-labeled nodes are $\perp$-parent.

(I2) Each parent is a A-labeled node.

(I3) The sub-graph induced by the $node\text{-}parent$ edges is a tree.

(I4) The obtained tree of the irreducible graph is a spanning tree.

Figure 18 gives a sample computation of a spanning tree with a crash of a process after the step T_4. The steps T_1, T_2 and T_3 represent the application of the main rule $RST1$. The process corresponding to the node shown by a star crashes after the step T_4, and remains in a faulty state until the end of the execution. Since the edge incident to this node belongs to the spanning tree (bold edge), it must be deleted from the tree and the adjacent node will be labeled N. That is done in step T_5 which is an application of RF_ST1 by the node in the square. Now, the latter node labeled N is a parent of a node labeled A. In step T_6, the node in the square applies the rule RF_ST2 by relabeling itself to N. Note that since RF_ST1 and RF_ST2 have highest priority, it will be applied on the context of the faulty node before $RST1$. Then, in step T_7, T_8, T_9, the rule $RST1$ is applied allowing to continue the computation of the spanning tree by avoiding the crashed node.

4.5 Computation of a Spanning Tree with Termination Detection

Let us illustrate fault-tolerant distributed algorithm which computes a spanning tree of a network with termination detection. We start with an algorithm in a network without crashes.

Assume that a unique given process called the "root" is in an "active" state (encoded by label (A,$\perp$)), all other processes being in some "neutral" state (label (N,0)). The tree initially contains the unique active node. At any step of the computation, an active node may activate one of its neutral neighbors. Then the neutral neighbor becomes active and marks with the variable par its activated neighbor. When node v_0 cannot activate any neighbor because all of them have already been activated by some other nodes, v_0 transforms its state into a "feedback" state. When all the activated nodes("sons") of v_0 are in the "feedback" state, it transforms its state into a "feedback" state. The root detects the termination of the algorithm when it is in the "feedback" state. Every process v_i maintains two variables:

- $span(v_i)$: is a variable which takes three values:
  - N: v_i is not yet in the tree
  - A: v_i is in the tree
  - F: v_i is in the feedback state, it finds all its neighbors in the tree and all its sons in the feedback state
  - T: the termination detection at the root
- $par(v_i)$: is the number of the port connected v_i to its activated neighbors

We consider the following relabeling system which encodes a distributed algorithm computing a spanning tree with termination detection, $\mathcal{R}_{f_2} = (\Sigma_{f_2}, \mathcal{I}_{f_2}, \mathcal{P}_{f_2})$ defined as $\Sigma_{f_2} = \{\{N, A, F, T\} \times \{1 \cdots deg(G) \cup \{\perp\}\}$, $\mathcal{I}_{f_2} = \{\{N\} \times \{\perp\}\}$, $\mathcal{P}_{f_2} = \{RSTT1, RSTT2, RSTT3, RSTT4\}$. Now we present the set of rules:

Algorithm 10 Computation of a spanning tree with termination detection

RSTT1 : **Root diffusion rule**

Precondition :
- $v_0 =$root
- $span(v_i) = N$

Relabeling :
- $span(v_0) := A$

RSTT2 : **Node diffusion rule**

Precondition :
- $span(v_0) = N$
- $\exists\, v_i \in B(v_0), span(v_i) = A$

Relabeling :
- $span(v_0) := A$
- $par(v_0) := v_i$

RSTT3 : **Node feedback rule**

Precondition :
- $v_0 \neq$ root
- $span(v_0) = A$
- $\forall\ v_i \in B(v_0) : (span(v_i) \neq N)\ and\ (par(v_i) \neq v_0\ or\ span(v_i) = F)$

Relabeling :
- $span(v_0) := F$

RSTT4 : **Root detection of termination rule**

Precondition :
- $v_0 =$ root
- $span(v_0) = A$
- $\forall\, v_i \in B(v_0) : span(v_i) = F$

Relabeling :
- $span(v_0) := T$

We present in the sequel a spanning tree with termination detection computed by a local fault-tolerant relabeling system. We start by defining some illegitimate configurations to construct a set $\mathcal{F}_{f_2}$, then we improve the system by adding the correction rules to detect and eliminate these configurations.

Note that we distinguish crashed node and faulty node as explained in the previous section. A faulty node should be viewed as one which has to reconstruct the computation because of the crash of some other nodes.

Definition 4.5.1 (correct node (faulty)) *A node v is correct (resp. faulty) if it satisfies one (resp. it satisfies none) of the following properties:*

1. *if $span(v) \in \{A, F, T\}$ and $par(v) = \perp$ then $v = root$,*
2. *if v is labeled (A, u) then there exists one node u labeled (A, w),*
3. *if v is labeled (F, u) then there exists one node u labeled (X, w), where $X \in \{A, F\}$,*
4. *if v is labeled $(N, \perp)$ then there does not exist node u labeled (X, v), where $X \in \{A, F\}$.*

For the present system, we deal with the following set $\mathcal{F}_{f_2} = \{f_1, f_2, f_3\}$ where f_1, f_2 and f_3 are:

1. $f_1 : \exists\, v_0 \neq$ root, $span(v_0) = A$ and $\neg\exists\, v_i \in B(v_0) : par(v_0) = v_i$ *and* $span(v_i) = A$.
2. $f_2 : \exists\, v_0 \neq$ root, $span(v_0) = F$ and $\neg\exists\, v_i \in B(v_0) : par(v_0) = v_i$ *and* $span(v_i) \in \{A, F\}$.
3. $f_3 : \exists\, v_0$, $span(v_0) \in \{A, F\}$, $par(v_0) = v_i$ and $span(v_i) = N$.

The correction rules are deduced from the previous configurations:

RF_STT1 : **Crash of a parent rule 1**

Precondition :
- $v_0 \neq$ root
- $span(v_0) = A$
- $\neg\exists\ v_i \in B(v_0) : par(v_0) = v_i$ *and* $span(v_i) = A$

Relabeling :
- $span(v_0) := N$
- $par(v_0) := \perp$

RF_STT2 : **Crash of a parent rule 2**

Precondition :
- $v_0 \neq$ root
- $span(v_0) = F$
- $\neg\exists\ v_i \in B(v_0) : par(v_0) = v_i$ *and* $span(v_i) \in \{A, F, T\}$

Relabeling :
- $span(v_0) := N$
- $par(v_0) := \perp$

RF_STT3 : **Cleaning rule**

Precondition :
- $span(v_0) \in \{A, F\}$
- $par(v_0) = v_i$
- $span(v_i) = N$

Relabeling :
- $span(v_0) := N$
- $par(v_0) := \perp$

We assume in this system that the "root" is usually correct. We define the relabeling system $\mathcal{R}_{f_2} = (\Sigma_{f_2}, \mathcal{P}_{f_2}, \mathcal{F}_{f_2})$, where $\Sigma_{f_2} = \{\{N, A, F, T\} \times \{1 \cdots deg(G) \cup \{\perp\}\}$ and $\mathcal{P}_{f_2} = \{RSTT1, RSTT2, RSTT3, RF\_STT1, RF\_STT2, RF\_STT3\}$ such that $RF\_STTj \succ RSTTi$. We now state the main result:

Theorem 4.5.2 *The relabeling system $\mathcal{R}_{f_2}$ is locally fault-tolerant. It encodes a fault-tolerant distributed algorithm to compute a spanning tree with termination detection.*

Proof. The proof of local fault-tolerant results from Theorem 4.4.2. To show that the result is a spanning tree, we use the following invariants:

(I1) All N-labeled nodes are $\perp$-parent.

(I2) Node v_0 which is labeled (A, v_i) has one neighbor v_i which is labeled (A, v_j) or $(A, \perp)$ if v_i is the root.

(I3) Node v_0 which is labeled (F, v_i) has its neighbor v_i labeled either (F, v_j) or (A, v_j). If v_i is the root, then v_i is labeled either $(A, \perp)$ or $(T, \perp)$.

(I4) Node v_0 which is labeled (F, v_i) has no (A, v_0)-labeled node as a neighbor.

(I5) The sub-graph induced by nodes v_i that are labeled (A, v_j), the root which is labeled $(A, \perp)$ and the node-parent edges has no cycle.

(I6) If G' is an irreducible graph obtained from graph G, then it contains exactly one $(T, \perp)$-labeled node and all others are labeled (F, v_i).

(I7) The obtained tree of the irreducible graph is a spanning tree.

4.6 Status and Future Works

As presented along this thesis, local computations model is a high level model to encode distributed computing systems. Algorithms encoded and proved in such a model give some intuitions to simplify their design in other models such as asynchronous message passing system. To capture and encode the concept of faults in this model, we introduce the $crash$ action. This fail step is used as an internal action. When some process v_f executes this action, it stops all its tasks. The effect of such changes could never be communicated to any other processes. The stop event is used only for formalization. Contrary, in the synchronous systems, the fail stop actions are detected by other processes. When we consider other models of computation, we must introduce some artificial mechanisms to avoid that the states of failure processes are communicated to other processes. That is, the resulting label after such action is not distinguishable by its neighbors or all other processes.

To circumvent impossibility implementation of fault-tolerant algorithms in asynchronous networks, there is a solution when we relax some constraints as states changes detection. In such a way we can introduce failure detector objects [CT96] in our formalization. Then, self-stabilization becomes fault-tolerance algorithm property. Along the motivation of our interest is the fact that few works consider the combination of failure detection service and self-stabilizing algorithms. Thus, we have presented a method to design fault-tolerant algorithms encoded by local computations. The method consists of specifying a set of illegitimate configurations to describe the faults that can occur during the computation, then adding local correction rules to the corresponding algorithm which is designed in a safe mode. These specific rules are of high priority and are performed in order to eliminate the faults that are detected locally. The proposed framework combines the failure detector implementation and the self-stabilization yielding a unified and simple framework to describe and study fault-tolerant algorithms. Hence, we can transform an intolerant algorithm into a local fault-tolerant one simply by using our developed framework. We demonstrate the feasibility of our approach via developing such algorithms for a distributed spanning tree algorithms in k-connected graph in the presence of at most $(k-1)$ crashed processes.

Our approach can be applied in practical applications as a generic and automatic method to deal with faults in distributed systems. Particularly, under an asynchronous message passing system which notifies the faults. For instance, in a very large network, assume that the

diffusion of messages between sites is performed using a spanning tree of the network. Now, if some central node crashes, then our method allows to find a solution to continue the diffusion service. We are currently working on applying our solution to particular architectures and mainly web services.

As introduced in the previous Chapter, in the future we will be interested to study the possibility to use heterogeneous failure detectors to design local fault-tolerant algorithms. The goal is to respond to the following question: *" how many failure detectors following their classes are necessary to solve such problems in unreliable networks ? "*. The important design corresponds to the minimum of weaker failure detectors combined to the minimum of higher failure detectors. For example, let $[1]$-Failure Detectors be a class characterized with the fact that each of failure detector may change its mind about some monitored process at most one. It means that its suspicion value may become $true$ and remains $true$ forever. The first step is how to implement such a protocol which is the expected work as given in Chapter 3. The second step is to study analytically or experimentally the number of failure detectors of such class required to solve some problem in spite of crash failures.

Chapter 5

A Tool to Prototype Fault-tolerant Distributed Algorithms

Development of distributed applications in dynamic networks in which nodes can fail, move or switch off is a hard task. In such environments the procedure requires many stages with independent requirements of each one. The protocols must be well specified and proved. The implementation that follows remains not trivial since often validation and debugging of a distributed system is a very complicated task. As distributed algorithms may involve a large amount of data encoding local state of components in the system and complex communications between them. In unreliable networks, the possible executions of algorithms increase and the involved processes and messages become larger over a period of time. Consequently, it is often very difficult (and sometimes impossible) to understand their control flow and their performance and behavior only from a code (relabeling systems or pseud-code) describing this kind of algorithms or from their execution traces. Thus, it is suitable to develop software which offer tools to deal with such applications. In particular, the use of visualization and animation to illuminate the key ideas (intuitions) of distributed protocols and algorithms.

In this chapter, we present a method to build an homogeneous and interactive visualization of fault-tolerant distributed algorithms including self-stabilizing algorithms encoded in the local computations model. The approach developed in this work unifies two levels. First, the formal framework presented in the both Chapters 2 and 4 as a high level to encode fault-tolerant distributed algorithms. Second, Visidia platform improved by the unreliable failure detector studied in Section 3.5.1 as a notification service. Therefore, this unification allows to automatically derive and prototype algorithms in distributed environments with failures. The resulting tool offers an interface to select some processes and change their states to simulate both transient and crash failures. Then, the animation allows to see the different possible executions of a distributed algorithm and failure detectors. This part of the thesis gives a set of techniques to demonstrate the functionalities of the algorithms and their behavior under an asynchronous distributed system model with failures. We illustrate the advantages of our tool by an example of a distributed algorithm to compute a spanning tree.

The rest of the chapter is organized as follows. In Section 5.1, we review some related works. The proposed method to deal with transient failures on the Visidia platform is given in Section 5.2. Then, we present in Section 5.3 the new version of Visidia platform devoted to

deal with crash failures. Section 5.4 gives an example of a possible implementation of a fault-tolerant spanning tree construction. Section 5.5 introduces an expected tool to help the design of distributed applications in a environment with crash failures. Finally Section 5.6 concludes the chapter with short discussions about extended works.

5.1 Related Works

Simulation and visualization of distributed algorithms are very important since their understanding and teaching are arduous. Several platforms have been proposed to help the designers of distributed applications [EH91, Sta95, FKTV99, KSV00, BHR+00]. They include debuggers, performance measurements and simulation runs.

In an alternative view, research efforts focus in the development of generic tools to visualize distributed algorithms execution [BHSZ95, Sta95]. Specifically, [BHSZ95] proposed a parallel programing language with a visual interface. Alternatively, in [Sta95] the motivations are to reveal concurrent programs and to provide a set of optional views corresponding to some specific aspects of the execution of the program (e.g. messages being transferred, local states, etc). Thus, the user monitors the program execution through selected views. Still, the case of execution containing a large amount of data. In this case, such information is given as a group of streams exploited via an animation program.

For sequential (centralized) programing, a very successful example is suggested in [MN99]. This software is composed of a library of data structures and algorithms for which is added visualization and drawing features. It is used as educational tool, as well as a prototyping tool in industry.

In [PT98], they described an integrated library devoted to education, such as for teaching distributed algorithms, communication protocols, and operating systems. For the protocols and the data objects implementation part, a simulation platforms for network and multiprocessor systems is implemented. To create network, a nice graphical editor was used based on a set of drawing algorithms. For the animation part, an appropriate and a powerful toolkit is used which supports many different architectures such as the simulation platforms. Recently, in [KPT03], they evaluated a distributed system assignment in which students used the platform of [PT98] to implement their algorithms. The feedback received gives valuable information about what the simulation-visualization environments must provide to be successfully taught and assimilated. Thus, the students are able to improve the performance of such a tool and help them to acknowledge the wealth of tools they are provided.

As stated before in Section 1.6, Visidia [MS, BMMS01, BM03] is a tool to implement, to simulate, to test and to visualize distributed algorithms encoded in both local computations (see Section 1.2) and message passing (see Section 1.3) models. The distributed system of Visidia is based on asynchronous message passing system. Figure 2 shows the components of Visidia. This tool is an unified framework for designing, implementing and visualizing distributed algorithms.

However, very few studies have shown conclusively that they are effective and may be used to deal with failures [Sta95, FKTV99, KNT05]. *Erlang* is a distributed functional language used to build reliable telecommunications systems. In [KNT05], they investigated the potential of this language to deliver robust distributed telecoms software. Its evaluation is based

on a typical non-trivial distributed telecoms application, a *Dis-patch Call Controller (DCC)*. Dispatch call processing is an important feature of many wireless communication systems. Since, the system components in such environments grows, and works are dynamically affected regardless of the resources available, it is usefulness to manage the call processing as distributed applications. The investigations of [KNT05] showed that the Erlang implementation meets the *DCC*'s resource reclamation and soft real-time requirements. The presented analysis concerns some reliability properties including those presented in Section 0.2.1. As a consequence, the fault tolerance model of Erlang is low cost, parameterizable and generic. Moreover, Erlang DCC is less than a quarter of the size of a similar C++/CORBA implementation, the product development in Erlang should be fast, and the code maintainable.

5.2 Simulation and Visualization of Self-stabilizing Distributed Algorithms

In this section, we present a method to build a homogeneous and interactive visualization of self-stabilizing distributed algorithms using the Visidia platform. This method allows a simulation of the transient failures and their corrective mechanisms. Local computations have been used to encode self-stabilizing algorithms such as the distributed algorithms implemented in Visidia. Although the resulting interface is able to select some processes and incorrectly change their states to show the transient failures, the system detects and corrects these failures by applying correction rules. Many self-stabilizing distributed algorithms are implemented and tested. We illustrate the advantages of our approach by two examples of distributed applications based on self-stabilizing algorithms.

The interface developed is based on the formal framework proposed in Chapter 2 [HM05a] and on the Visidia platform. The graph relabeling system encoding the self-stabilizing distributed algorithm is implemented by high level primitives of Visidia. An interface allows the user to change the state of the processes to simulate transient failures. In [HM05c], we illustrated our methodology by examples of spanning tree computation and an algorithm to propagate information with acknowledgment. Then, we apply the later algorithm [HM], often called a PIF algorithm, to design applications in distributed environment with transient failures. We use such an algorithm as a basic protocol to implement both an automatic mail system to broadcast a lot of messages and a maintaining of a common data system for distributed sites. Thus, our approach can be applied in practical applications as a generic and automatic method to deal with transient failures in distributed systems.

5.2.1 Self-stabilizing Algorithms on Visidia

The start-up and the configuration of the simulation is achieved using an interactive interface. The designer may change the labels, and then the data of the specified process. Then, the new values are applied to the data of such a process. It is possible to visualize the content of such data by means of display views. A break-point ability of the execution of the algorithm is available.

In the presence of transients failures, the self-stabilizing property of the designed application allows to maintain correct behaviors. The user can simulate a transient failure of a process with the graphical user interface before the start of the simulation or during the simulation.

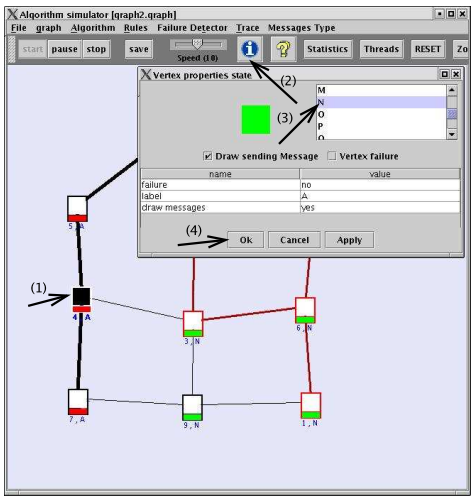


Figure 19: Error introduced by the user to simulate a transient failure

Using the mouse through the graphical user interface, the designer chooses a process by selecting its corresponding node and changes its status to a *corrupted* one:

- By starting the execution of the algorithm with a randomized value of the variables of the program (labels). This is possible because a local stabilizing graph relabeling system can have any initial configuration.
- By using the mouse through a graphical user interface. After the suspension of the execution the user can simulate the failure of a process by selecting a node of the graph representing the network. In Figure 19, at step (1) the user selects the node in which the transient fault will be introduced. In step (2), the user opens the view to change the state of the selected node. Step (3) changes the state and step (4) validates this operation. Then, it may continue the suspended execution.

5.2.2 Examples of Applications

There are many applications that are based on the knowledge of a special structure of graphs. For the routing applications [CW04] each node maintains some subset of the network topology and uses this subset to perform adequate routing. An important measure in these routing schemes is the number of nodes to be updated upon a topology change. So, it is necessary to collect information about the network state, e.g., nodes and links operational status, to update the routing tables. It is suitable to investigate the usefulness of maintenance of a special structure in the design of efficient routing schemes.

Since data and the status of processes are encoded using labels, properties involving the values of some data, such as counters for example, can now be checked. Above it's only checked in theory because the number of possible configurations and then local states is very prohibitive. In the presence of failures, the tool takes into account the correction behaviors envisaged during the design of algorithms. Then, the corresponding scenario will be able to be replayed immediately at the simulator screen. Thus, the debugging time is significantly reduced and the proposed interface allows a constructive methodology with the designers.

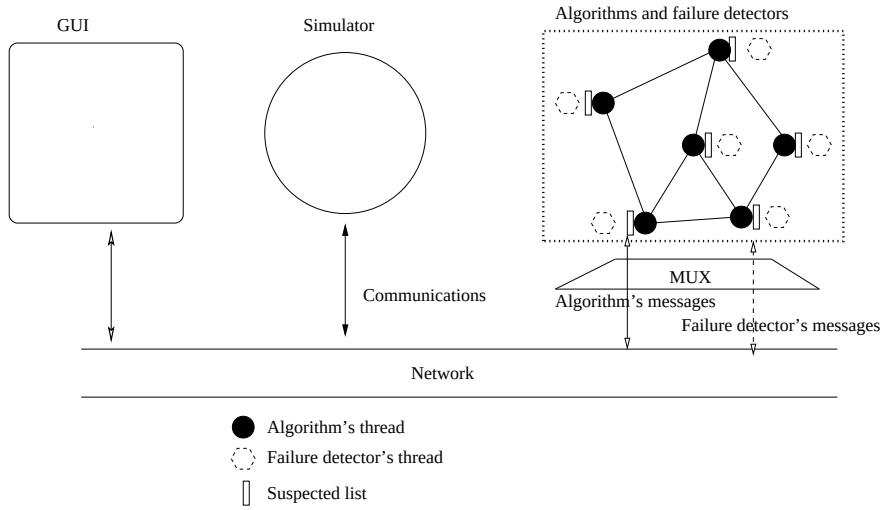


Figure 20: Visidia architecture improved by failure detectors

Broadcast with feedback scheme has been used extensively in distributed computing to solve a wide class of problems, e.g., spanning tree construction, snapshot, termination detection, and synchronization. In [HM], we presented two possible applications of the self-stabilizing PIF algorithm.

5.3 Visidia Platform to Simulate Fault-tolerant Distributed Algorithms

Here we present our methodology devoted to help the designing of fault-tolerant distributed applications. Our technique is based on the use of failure detector objects on each node to ensure failure detection service. The unreliability of such services is completed by some correction rules executed on each node to correct themselves as proposed in the previous Chapter. Correction rules are defined according to suitable behaviors of the algorithms themselves.

Each process of the distributed application to be simulated has an associated "Visidia algorithm thread", "Visidia failure detector thread" and "list of suspected" object placed between the two previous "threads" as shown in Figure 20.

5.3.1 Architecture of Visidia

The communications primitives of Visidia are improved adding sending and receiving failure detections messages. Moreover, the existing primitives are extended to tack into account the waiting delay. Some subroutines are also defined to implement the failure detection notification service. Recall from Chapter 3 and Chapter 4 that the synchrony is required for the failure detector implementation and the algorithm is still implemented in asynchronous mode. Thus, we distinguish between the failure detector communication mechanisms and those of the algorithm. In the following we present a short description of the main added and modified subroutines:

1. $receiveFrom(intdoor, longtimeOut)$:

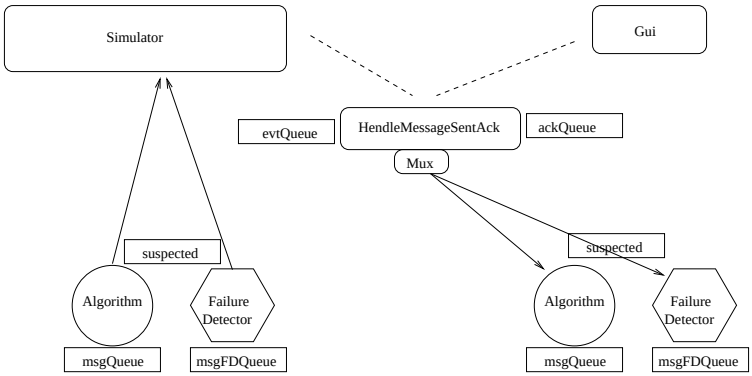


Figure 21: Visidia Platform to simulate fault-tolerant distributed algorithms

2. $suspected(intdoor)$:
3. $suspectedId(intid)$:
4. $existFDMessage()$:
5. $existFDMessageFrom(intdoor)$:
6. $getState(intidautre)$:
7. $putState(intidautre, intsusp, intcount)$:
8. $getSuspected()$:

As shown in Figure 20 and more precisely in Figure 21, the simulator component multiplexes messages coming from the different "threads" based on the message type [1]. All of the "algorithm", the "failure detector" and the "suspected list" have an associated objects in the three components or layers of Visidia.

In the following, we present an implementation of an unreliable failure detector based on the "heartbeat" mechanism. Then, we show the main contribution of this chapter.

5.3.2 Implementation of a Failure Detector

Recall from Chapter 3, the method proposed to detect failures in the local computations model and its corresponding implementation in the message passing model. The failure detector module is implemented by a *Java* thread to execute Algorithm 7. As shown in Figure 20, the main algorithm is executed by the process computing unit, and that the detector module is executed independently. However, the suspected lists maintained by the failure detector are accessible by both. Each failure detector communicates with the simulator which transfers the events to the visualizer. The simulator uses a filter to distinguish between failure detectors messages and those of algorithms as explained above.

The start-up and the configuration of the failure detector is achieved using an interactive interface. The designer may change the parameters of the specified failure detector, including the "heartbeat" interval, the "waiting delay" and the failure detector messages. Then, the parameters are applied to all the failure detectors. Remember that the only task achieved by

[1] At the reception of messages, the simulator demultiplexes them.

the failure detector is the maintaining of its corresponding suspected list. It is possible to visualize the content of such lists by means of display views. A break-point capability of the failure detector execution is available such as that of the algorithm execution.

5.3.3 Implementation of Fault-tolerant Distributed Algorithms

To implement fault-tolerant algorithms, Visidia platform is improved by an unreliable failure detector to assure the suspicion service of the crashed processes. Thereafter, if the system is notified by the failure detector that some processes crashed, the correction rules are applied. Whenever a $receiveFrom$ action is invoked by some process, it waits until the reception of a message. To avoid the "deadlock waiting", each $receiveFrom(u)$ action executed by every process v will be replaced by action of the format: $receiveFrom(u)\ \ OR\ \ u \in Suspected(v)$. Thus, when Visidia is used to simulate unreliable applications, the receiving actions will be implemented using Algorithm 11. So, the action $receiveFromFree$ and the set $Suspected(v)$ are added to Visidia. The latter denotes the set of processes that are suspected by v.

Assume that each local computation and its needed synchronizations are atomic steps. It means that failures occur before the start of a local computation or after its ending (see Algorithm 12). The synchronization procedures are assumed to be failure free. We will base our methodology on the suspected lists exchanged between correct processes and on the correction rules introduced in the algorithms. The service guaranteed by the failure detector is unreliable, therefore the algorithms must correct themselves. Synchronization procedures and local computations will use suspected lists via the function $suspected(door)$. It returns $true$ if the neighbor corresponding to the port $door$ is suspected and $false$ otherwise. This function implements an "internal communication" service between the failure detector [2] object and the algorithm[3].

Algorithm 11 receiveFromFree

```
 msg = receiveFrom(door,TIME_OUT_FIRST);
 while ((!suspected(door))&&(msg==null)){
    msg = receiveFrom(door,TIME_OUT_SUSP);
 }
return msg;
```

In the presence of crash failures, the fault-tolerant property of the designed algorithms allows to maintain correct behaviors. The user can simulate the crash failure of a process with the graphical user interface before the start of the simulation or during the simulation. Using the mouse through the graphical user interface, the designer chooses a process by selecting its corresponding node and changes its status to *crash* one. After this step, both threads of the algorithm and of the failure detector corresponding to this node are halted as shown in the right part of Figure 22. In the example given in Algorithm 12, the sub-tasks "synchronizationFree()" and "computationFree()" denotes respectively a synchronization procedure and a local computation. Both of these sub-tasks use Algorithm 11 to implement their communication actions.

[2] $Read \setminus Write$.
[3] $Read$.

Algorithm 12 Example of execution of one rule

```
while (run) {
  //start of an atomic step
  synchronizationFree() ; // start of a synchronization
  computationFree();
  breakSynchro(); //end of a synchronization
  //end of an atomic step
}
```

5.4 Example of Distributed Computation of a Spanning tree

Now, we show the power of the presented methodology with its use to simulate distributed algorithms on unreliable networks. We deal with an example of a spanning tree computation.

We consider a graph relabeling system to encode a distributed computation of a spanning tree using labeled function L. Each node u is labeled using three components ($L(u) = (span(u), par(u), Susp(u))$). First, $span(u) \in \{N, A\}$ to mean that u is included in the on-building tree (label A) or is not yet in the tree (label N). Second, $par(u) \in \{1 \cdots deg(u)\} \cup \{\perp\}$ to denote the number of the process which activated u, or simply the port number linking u to this node. We say that v is a parent of u and u is a son of v. Third, the set $Susp(u)$ composed of neighbors suspected by u. When a node detects the faulty of its neighbors, it applies corresponding correction rules to become in a correct state. The spanning tree is defined using marked edges. As the graph required must remain connected during the whole execution, in the $k - connected$ graph at most $(k-1)$ failing processes at the same time is permitted [4].

Algorithm 13 Complete implementation of the local fault-tolerant distributed algorithm

R1 : **Node v includes itself in the tree**

Precondition :
- $span(u) = N$
- $\exists\, v \in B(u),\ span(v) = A$

Relabeling :
- $span(u) := A$
- $par(u) := v$
- $L(u, v) := 1$ /* visualization of the edges of the tree */

RC1 : **Node u suspects one of its sons**

Precondition :
- $span(u) = A$
- $\exists\, v \in B(u),\ par(v) = u,\ v \in \mathbf{Susp(u)}$ and $L(u, v) = 1$ /* the list of suspected processes may be shown via views */

Relabeling :
- $L(u, v) := 0$

RC2 : **Node v suspects its parent**

Precondition :
- $span(u) = A$
- $\exists\, v \in B(u),\ par(v) = u$ and $v \in \mathbf{Susp(u)}$

Relabeling :
- $span(u) := N$
- $par(u) := 0$

RC3 : **Node v has an incorrect state**

Precondition :
- $span(u) = A$
- $par(u) = u$ and $span(v) = N$

Relabeling :
- $span(u) := N$
- $par(u) := 0$

Now, we investigate an execution of the previous algorithm on our tool. Left part of Figure 22 exhibits some state of the computation without crashes. As mentioned above, status, labels and messages exchanged between processes and between failure detectors are displayed on the screen. Using the interface, process of node 3 has been set in the crash status as shown in the right part of Figure 22. Thereafter, failure detectors corresponding to the neighbors of the

[4] A graph admits a spanning tree iff it is connected.

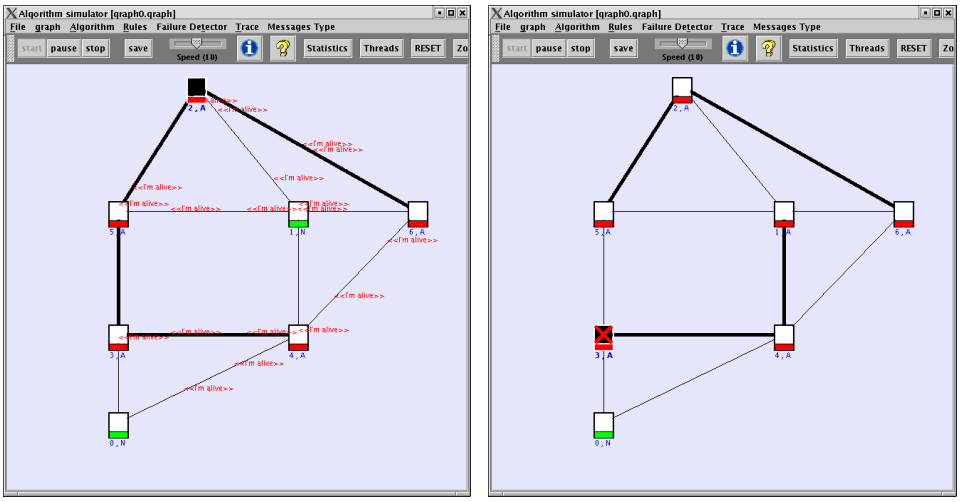


Figure 22: Error introduced by the user to simulate a crash failure

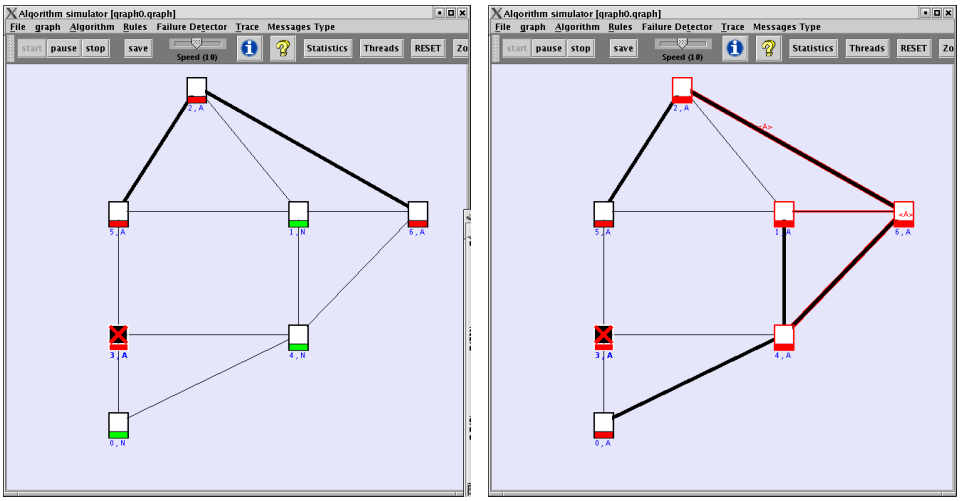


Figure 23: Execution of corrections rules until spanning tree is computed

crashed node eventually detects this status. Thus, neighbors of node 3 apply adequate correction rules to correct themselves. Left part of Figure 22 illustrates the execution of correction rule $RC1$ by node 5 and correction rule $RC2$ by node 4. Correction rules are executed until the correction of all the infected part in the graph. In the right part of Figure 22, we have a new computation of a spanning tree without the crashed node.

5.5 Fault-tolerant Distributed Applications

This section focuses on the design and validation of reliable applications in distributed environments. Our interests include the simulation of the hardware and software architecture of the system, of the communication infrastructure in such environments, and the behavior of algorithms in order to determine and measure the dependability impact of various failure scenarios, identify critical components, and suggest improvements.

Our goal is to help the design and implementation of software environments to provide

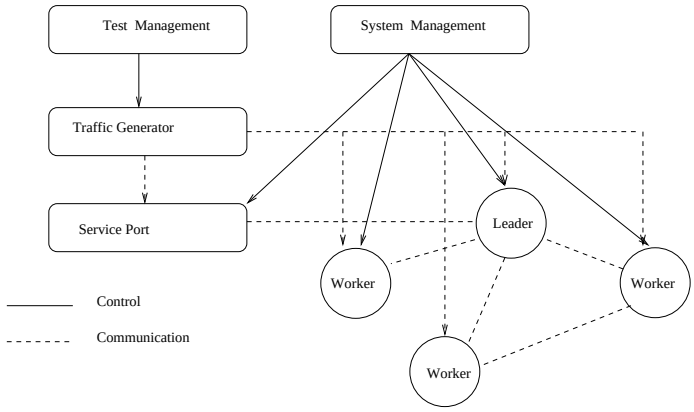


Figure 24: System architecture

adaptive levels of fault tolerance. The proposed approach address methods for ensuring predictable dependability and responsiveness in network environments, including both homogeneous and heterogeneous systems components. Validation of protocols in such approach includes the tests of high-performance, networked configurations that must deliver high-availability services under heterogeneous operating systems, computer platforms, and switching technologies.

We believe that system architectures such as shown in Figure 24 (for the *DCC* [KNT05]) are treatable in Visidia platform. We will illustrate one of the possible deployment. For the creation of the network we proceed in two phases: (1) The set of workers is described as usual nodes, (2) and the other components are described also in the same way using different graphical representations. The encoding of the corresponding protocols is similar to the example studied in the previous section except here we add some control layers to allow the execution of the different part of the algorithms simultaneously.

5.6 Status and Future Works

Simulation and visualization of distributed algorithms are very important since their understanding and teaching are arduous. That is, seeing that each execution of such algorithms involves many processes and a large quantity of messages. In fact, animation of a distributed algorithm graphically shows one of its possible executions. Therefore, such interfaces make easier its understanding and its study. In this way, software visualization systems seem useful to teach and to software development.

In this chapter, we have presented a powerful method to build an homogeneous interactive visualization of fault-tolerant distributed algorithms. The resulting interface offers views to select some processes and change their states to simulate the failures. This method will help the study and the complexity analysis of such algorithms.

For the transient failures, the resulting interface is able to select some processes and incorrectly change their states to simulate such failures. Then, the system detects and corrects these failures by applying correction rules as presented in the formal framework. The tool benefits from the support of formal methods, that allow the automation of generation of self-stabilizing distributed applications. The unification, between the formal model to encode self-stabilizing

applications and the visualization approach, presented here may be extended to be used as a previous phase to design self-repairing and self-organizing distributed applications. The simple examples described in this work show its capacity to develop a specific protocol to encode a certain behavior of distributed applications, to maintain such a behavior and recover in a finite time to this behavior in the presence of transient failures.

For the crash failures, the platform is improved by unreliable failure detector objects. Such objects implement a service to notifies the system when it detects failures. After that, the execution of the algorithm in the nearest part of the crashed processes is restricted to the application of the corresponding correction rules. Such rules are applied until the elimination of all the incorrect behaviors.

The center of interest of Visidia is the design of prototypes. So, this tool helps designers to do some measurements in terms of messages exchanged and to understand the all possible executions of the distributed algorithms. Consequently, this tool may be used for verifying and for proving the correctness of distributed algorithms. These verifications are particularly useful during both the dimensioning of the network and the development of new protocols interacting with protocols already developed. The proof of the expected properties of the prototyped protocols, using our tool, offers a powerful mechanism to validate such applications. Many real networks are based on the use of "Java sockets", the communications are implemented using "IP" addresses and the port number of the processes involved in the computation. It's possible to use our tool to simulate an unreliable message delivery protocols. Moreover, many parameters related to the used protocols on the specified networks may be experimentally computed as "the round-trip time". We believe that the result of this chapter is of practical interest. Basically, our framework and the resulted platform, aid to illuminate the key ideas to augment asynchronous systems with appropriate failure detectors.

As well-known, distributed algorithms can be difficult to understand as well as to teach. Thus, we hope that Visidia will very helpful for educational purposes, such as for teaching fault-tolerance in distributed computing, in communication protocols, in operating systems. For example, it is easy to show that different execution of the same algorithm, even if they start from the same initial system configuration, may not result in the same result, due to the synchrony (in-determinism). Since the most part of the implementation of Visidia was achieved with students, they are able to improve the performance of such tool and help them to acknowledge the wealth of tools they are provided. Still, we expect additional educational benefits from Visidia in the future.

Now, we are interested in extending our approach to deal with problems related to real networks including web services. More generally, the well-know strategy to design reliable systems, composed of multiple components, is the replication of its all or some components. The increasing of the replication degree improves the reliability of the system but the cost of maintaining such replications is expensive. As implicitly mentioned in this work, we preferred an autonomous and local correction of failures after their detection. In this case we assume that corruptions infect data that are generated using usual correct programs. On other hand, when the exchanged data include part of program it's useful to study the behavior of the validated protocols in such environments. We will add to Visidia an interface to select the desired code program to execute, and the possibility to select corrupted version of such programs. We expect that these improvements allow us to understand and to design easily fault-tolerant data and code distributed applications. Thus, the costs of such implementations will be significantly minimized.

We plan also to study the possibility to use our platform to present an easier way to understand and to prove some impossibility computations. For example, an alternative proof of the well known consensus problem [FLP85] using our platform may be done using the following: We consider a simple star network composed of a specified node and a set of neighbors. All nodes are initialized with some labels from the set $\{0, 1\}$. The goal of such an application is to choose the same value from the initialization values using only exchange of messages without any assumption about the message transfer delay. We have two configurations: First, without using failure detection service and second we run the same application using a previous failure detection service. The platform allows us to modulate globally the failure detector parameters. The parameters are used to regulate and to measure the performances of such a service. Running experiments with some process failure under the first configuration shows that the application does not terminate. The number of exchanged messages exploded and the used machine is not able to do this task. Under the second configuration, the choice of the failure detector parameters needed to terminate the computation in a correct manner is a hard or an impossible task.

Chapter 6

Distributed Local Test of 2-Vertex Connectivity

Vertex connectivity of a graph is the smallest number of vertices whose deletion separates the graph or makes it trivial. In general, graphs offer a suitable abstraction for problems in many areas. For instance, when a communication network is represented as a graph, the measure of its connection is the same as the k-connectivity of the graph. An undirected graph is connected if there is a path between every pair of its nodes. In fact, the quality of the reachability of any couple of nodes in an unreliable network, and hence their communication, will depend on the number of paths between these nodes. The number of disjoint paths is the connectivity. Since it has numerous applications in dynamic and unreliable networks, distributed connectivity test of a graph has gained much attention recently. Indeed, many distributed applications such as routing or message diffusion assume that the underlying network is connected [BHM02, CW04, Tel00, Wes01]. Connectivity information is also useful as a measure of network reliability. This information can be used to design distributed applications on unreliable networks. For example, testing and then preserving k-connectivity of a wireless network is a desirable property to tolerate failures and avoid network partition [BHM02].

Here, we investigate the problem of the test of the 2-vertex connectivity of a graph $G,$ or simply 2-connectivity. That is, we deal with the problem of testing whether G is 2-connected or not. A protocol that solves such a problem responds to the question: The number of nodes that must be deleted in order to disconnect a graph is at least 2. Furthermore, we study this problem in distributed setting using only local knowledge. This means that each node "communicates", or exchanges information, only between neighboring nodes and each computation acts on the states of neighbors. In this chapter, we propose a general and a constructive approach based on local knowledge to test whether a given graph modeling a network is 2-connected in a distributed environment. Therefore, our work may be used as a core of protocols using the connectivity as input information.

Now, we present briefly the main idea of our approach. Let G be a graph with a distinguished node v_0 and let T be a spanning tree of G rooted at v_0. It is known that for a graph with a distinguished node, a distributed computation of a spanning tree can be easily performed [MMS02]. Therefore, the assumption of a pre-constructed spanning tree is not a loss of generality. The vertex connectivity test algorithms presented here have the following

characteristics: Each of them is based on a pre-constructed spanning tree and uses only local knowledge. That is, to perform a computing step, only information related to the states of the neighbors is required. Especially, each node knows its neighbors in G and in the current spanning tree T the "position" of a node is done by its $father$ except for the root, and a set of ordered $sons$. The algorithms make a traversal of the tree starting from the root. Each visited node v computes the set of its sons not reached by the root in the graph G deprived of v. So each node checks if all its sons are reached by the root in the sub-graph induced by its deletion. The graph is 2-connected iff all the checks are positive. In the opposite case, the graph is not 2-connected. Our algorithms work without any more assumption for any node except the root. Moreover, this protocol does not need a completely identified network. Only the root needs to be identified: The network is *semi-anonymous*. To take into account the particular case of the root, one of its son is chosen to check if all the other sons are reached by such a son in the graph deprived of the root.

The rest of the chapter is organized as follows. In Section 6.1, we review most related works. The distributed system model, including the notion of procedure in our models using example of a spanning tree computation, is explained in Section 6.2. Then, we present our approach to design formal procedures dealing with the problem of the 2-vertex connectivity test of graphs in Section 6.3. Section 6.4 is devoted to the encoding of our protocol in the local computations model. The proof of its correctness and its complexity analysis are also given. In Section 6.5 we propose an implementation of this protocol in the asynchronous message passing model. We show that the proofs and the complexities analysis may be deduced from the previous encoding using simple changes. Then in Section 6.6 we conclude the chapter and we discuss some of its possible extensions.

6.1 Related Works

In the late 1920s, Menger [Wes01] studied the connectivity problem and some related properties. Since then many results have been obtained in this area. The best known works about the k-vertex connectivity test problem may be summarized in [Wes01, Tar72, ET75, EH84, HRG00, Gab00]. The above list of references is not complete.

Given a graph $G = (V, E)$, we denote by N the size of the node set and by M the number of edges. The maximum degree of G is denoted by Δ. In [Tar72] an algorithm to test the 2-connectivity of graphs is given. This is a *depth first search* based algorithm with a time complexity of $O(M + N)$. Thereafter, the computation of the vertex connectivity is reduced to solve a number of max-flow problems. For the first time this approach was applied in [ET75]. The computation of the maximum flow is used as a basic procedure to test the vertex connectivity. The above algorithm makes $O(k(N - \Delta - 1))$ calls to max-flow procedure.

The remaining works try to reduce the number of calls to max-flow procedure using some knowledge about the structure of the graph. The time complexity of the presented algorithms is bounded by $O(k \times N \times M)$.

The work of [EH84] reduces the number of calls to max-flow procedure using some knowledge about the structure of the graph. The number of calls the algorithm requires, is bounded by $O(N - \Delta - 1 + \frac{\Delta(\Delta-1)}{2})$. The algorithm given in [HRG00] to compute the vertex connectivity k, combines two previous vertex connectivity algorithms [ET75, GGT89] and a generalization of the preflow-push of [HO94]. The time required for a digraph is $O(min\{k^3 + n, kN\}M)$;

for an undirected graph the term M can be replaced by kN. A randomized version is also presented which finds k with error probability $1/2$ in time $O(NM)$. The approach of [Gab00] uses expander graphs to find vertex connectivity. This structure allows to exploit nesting properties of certain separation triples. The time complexity of the presented algorithm is $O((n + min\{k^{5/2}, kN^{3/2}\})M)$ for digraphs and replacing M by kN for undirected graphs.

Note, however, that all of these works are presented in a centralized setting assuming global knowledge about the graph to be examined.

6.2 The System Model

The distributed system (network) is modeled by a graph $G = (V, E)$ model. Thus the connectivity of the network means the connectivity of the graph modeling it. The network is modeled by a graph $G = (V, E)$, where V is the set of processes. An edge (u, v) in E stands for an unidirectional link from u to v. Processes communicate and synchronize by sending and receiving messages through the links. Processes and links are assumed to be reliable.

To encode distributed algorithms, we consider both graph relabeling systems model and asynchronous message passing model. For the first one, we use only relabeling between two neighbors. That is, each one of them may change its label according to rules depending only on its own label and the label of its neighbor. We assume that each node distinguishes its neighbors and knows their labels. In the sequel we use the set $B(v)$ to denote the set of (locally) ordered immediate neighbors of v which is an input data. For the second one, each process (node of the graph) has its own local memory, with no shared global memory. Processes communicate by sending and receiving messages through existing communication links (edges of the graph). Networks are asynchronous in the sense that processes operate at arbitrary rates and messages transfer delay are unbounded, unpredictable but finite. However, we assume that messages order are preserved. Furthermore, the network is *semi-anonymous*: Only the root needs to be identified.

Here, the term "local computations model" is used instead of "graph relabeling systems model". For a sake of clarity, we will use "encoding" (resp. "implementation") when we use the local computations model (resp. the message passing model) to encode distributed algorithms. In both models, we use the term label (and sometimes variable), attached to each node or process, instead of the own data manipulated by each of the program at this process.

As we will see, some of the algorithms presented in this chapter will be used later in the rest of the thesis as building blocks for other, more complex, algorithms. Thus, we introduce the notion of procedure and we illustrate this notion with an example of a spanning tree computation in both these models, see the next subsection.

In this work we use the following conventions and notations:

1. To encode information stored at each node, we use labels (resp. variables) in the local computations model (resp. in the message passing model).

2. The steps executed by each node are described as rules (resp. actions) in the local computations model (resp. in the message passing model).

3. The rules are tagged by R (resp. A) in the local computations model (resp. in the message passing model).

4. The names of algorithms are suffixed by GRS (resp. MPS in the local computations model (resp. in the message passing model).

For a sake of clarity, in the explanation of the algorithms encoded in the message passing model we refer to a node instead of its corresponding port number. Note, however, that in our solution the identities of nodes are not required.

We propose the following fact and definitions that we will use in the following:

Fact 6.2.1 *Let T be a spanning tree of G with v_0 as root. Let $v'_0 \in V_T$ and let $p = (v_0, v_1, \cdots v_l)$ be the simple path in T where $v_l = v'_0$ then the tree T' with v'_0 as root, associates with the partial order $\overline{\leq}$ defined by :*

- $\forall v, v' \notin \{v_0, v_1, \cdots v_l\}, v \leq v' \iff v \overline{\leq} v'$
- $\forall v, v' \in \{v_0, v_1, \cdots v_l\}, v \leq v' \iff v' \overline{\leq} v$

is also a spanning tree of G.

Definitions 6.2.2

- *We denote by T_r a tree with root r. Formally, $\forall\, v \in V_{T_r},\ v \leq r$.*
- *We denote by $T(v) \setminus r$ the maximal sub-tree of T that contains the node v, but not r.*

Definition 6.2.3 *A vertex v of a connected graph G is a cut-node iff $G \setminus v$ is not connected.*

Correction and Termination Proofs To make easier the understanding of the run and the power of the presented algorithms, we use operational proofs rather than formal proofs. Note, however, that the formal proofs may be done as the *spanning tree procedure* presented in Section 1.2.6 for example. For the proofs of termination, here they are encapsulated in the complexity analysis. Similarly, they can achieved using the formal technique introduced in Section 1.2.

On another hand, the correction and termination proofs of an algorithm composed of a set of procedures is based on those of its procedures. That is, we assume that when the procedures are executed in the relabeling part of some rule, they are executed as atomic steps with the two following consequences:

- Each procedure achieves its expected task,
- Each procedure terminates during the execution of the rule referencing it.

6.2.1 Spanning Tree Procedure

We present here a distributed algorithm to compute a spanning tree T of a graph G [Tel00, MMWG01], encoded in both the local computations and the message passing models. Since it will be used by our algorithm, it is presented as a "procedure" with parameters. This notion of procedure is similar to the "interacting components" used in [PLL97]. The *spanning tree procedure* is referenced as follows:
$\mathbf{STP\_GRS}(\mathbf{G}, \mathbf{T}; \mathbf{v_0}; \mathbf{Stage}, \mathbf{X}, \mathbf{Y}; \mathbf{Father}; \mathbf{Sons})$, where G is the treated graph and the procedure builds a spanning tree T rooted at v_0. The structure of T is stored locally in each node using the labels $Father$ and $Sons$ described previously. Initially, all the nodes v have

their labels $Stage$ valued to X. At the end of the computation, each node v will be labeled $Stage(v) = Y$. The following graph relabeling system describes an encoding of the *spanning tree procedure*.

Algorithm 14 Spanning tree procedure ($STP\_GRS(G, T; v_0; Stage, X, Y; Father; Sons)$)

- **Input :** A graph $G = (V, E)$ and v_0 the chosen root.
  - **Labels:**
    - $B(v)$: the set of (locally) ordered immediate neighbors of v which is an initial data,
    - $Stage(v)$: the state of v can take many values. The only used in this procedure is: "X" as the initial value, "WA" to mean that v has been included in the tree and "Y" to mean that v finished locally its computation.
    - $Father(v)$: is the father of v in the on-building spanning tree.
    - $Sons(v)$: an ordered list of the sons of v in the previous tree.
    - $Included(v)$: the set of v's neighbors included for the first time in the on-building spanning tree.
    - $Terminated(v)$: the set of v's neighbors that finished locally the computation of the spanning tree.
  - **Initialization:**
    - $\forall v \in V,\ Stage(v) = X$.
- **Results:** A spanning tree $T = (V, E_T)$ of G with root v_0 such that $\forall v \in V,\ Stage(v) = Y$.
- **Rules:**

$STR1$: **The node v_0 starts the computation**

Precondition :
- $Stage(v_0) = X$

Relabeling :
- $Stage(v_0) := WA$
- $Father(v_0) := \perp$
- $Sons(v_0) := \emptyset$
- $Included(v_0) := \emptyset$
- $Terminated(v_0) := \emptyset$

$STR2$: **Spanning rule acting on 2 nodes v, w where w is not yet in the on-building tree**

Precondition :
- $Stage(w) \neq WA$
- $v \in B(w)$ and $Stage(v) = WA$

Relabeling :
- $Stage(w) := WA$
- $Father(w) := v$
- $Sons(w) := \emptyset$
- $Terminated(w) := \emptyset$
- $Sons(v) := Sons(v) \cup \{w\}$
- $Included(v) := Included(v) \cup \{w\}$
- $Included(w) := \{v\}$

$STR3$: **Node v discovers its neighbors already included in the tree**

Precondition :
- $Stage(v) = WA$
- $w \in B(v)$, $Stage(w) = WA$ and $w \notin Included(v)$

Relabeling :
- $Included(v) := Included(v) \cup \{w\}$
- $Included(w) := Included(w) \cup \{v\}$

$STR4$: **Node v finishes locally the computation of a spanning tree**

Precondition :
- $Stage(v) = WA$
- $Included(v) = B(v)$ and $Terminated(v) = Sons(v)$

Relabeling :
- $Stage(v) := Y$
- **if** $(Father(v) \neq \perp) Terminated(Father(v)) := Terminated(Father(v)) \cup \{v\}$

Here, $Father(v) = \perp$ means that v has no defined father. First, v_0 the root of T applies the rule $STR1$ to initialize the computation. At any computation step based on the application of the rule $STR2$, when a node w not yet in the tree, finds a neighbor v in T labeled $Stage(v) = WA$, this node (w) includes itself in the tree by changing its $Stage$ to WA. Moreover, at the same time, $Father(w)$ is set to v and w is added to the sons list of v. So at each execution of the rule $STR2$, the number of nodes not yet in the tree decreases by 1. When v finds all its neighbors already included in the tree (rule $STR3$), it applies the rule $STR4$. This means that v has locally terminated its computation, then it informs its father. Note that this rule is executed firstly by the leaves. The computation finishes when $Stage(v_0) = Y$. In this case, all the nodes v also satisfy $Stage(v) = Y$. Obviously we have a spanning tree of G rooted at v_0 defined by the third components and the fourth components of the labels of the nodes. The root of the spanning tree is then the unique node with its father equal to $\perp$. Therefore, we claim the following fact:

Fact 6.2.4

1. *The rule* $STR1$ *is applied once,*
2. *The rule* $STR2$ *is applied* $\#V - 1$ *times,*
3. *The rule* $STR3$ *is applied* $\#E - (V - 1)$ *times,*
4. *The rule* $STR4$ *is applied* $\#V$ *times.*

Then, the following property states the effect and the cost of the execution of the *spanning tree procedure* on a graph G to build a spanning tree T rooted at v_0 where initial labels are X and resulting labels are Y.

Property 6.2.5 *For a graph* $G = (V, E)$, *when a distinguished node* v_0 *is labeled* $Stage(v_0) = Y$, $(\#E + \#V + 1)$ *rules have been applied. Then, the spanning tree* T *of* G *rooted at* v_0 *has been built.*

In the sequel, we will need to define some procedures to encode our algorithms. For the sake of uniformity, procedures will use the following standard header format.

$$\mathbf{name}\ (\mathbf{struct_0}, \cdots \mathbf{struct_i}; \mathbf{node_0}, \cdots \mathbf{node_j}; \mathbf{lab_0}, \mathbf{val_0}, \mathbf{val'_0}; \cdots ; \mathbf{lab_l}, \mathbf{val_l}, \mathbf{val'_l})$$

The header of the procedure is composed of its name and a set of optional parameters. Each of the sets is separated using the character ";". The first set of parameters is the structures of the manipulated graphs, the second is the set of the distinguished nodes. The rest is related to used labels, their required initialization values and their expected values.

To implement procedures in the local computations model, each of them is composed of three parts. First, the input part to describe the set of the used labels and their required initialization. Second, the result expected after the execution of such a procedure in terms of labels. Finally, the set of relabeling rules to specify the task of such a procedure. With each procedure, we associate some properties, maintained using invariants, to prove the correctness, to show the behavior and to measure the complexity of such a procedure. Invariants may be proved using induction on the size of the relabeling sequences as given in [LMS99]. Therefore, the correctness of an algorithm using such procedures results from those of these procedures. The complexity is based on those of the procedures and the number of their invocations in the designed application. Since procedures are executed as atomic actions in the relabeling part of the rules, the power of local computations model is not altered. Atomic execution of procedures is possible in our local computations model since the sub-graphs on which the procedures are applied simultaneously are disjoint.

To make easier the readability of the analysis, we define the function $COST(\mathcal{A}, \alpha)$ to give the cost of the application of the given procedure (or algorithm) $\mathcal{A}$ in terms of complexity specified by the parameter α. For example, for a given graph $G = (V, E)$:
$COST(STP\_GRS(G, T; v_0; Stage, X, Y; Father; Sons), time) = \#E + \#V + 1$.

As depicted in Algorithm 15, the same extension is added to the message passing model: Here we give an example of the previous algorithm implemented in this model.

Here the variable $Father$ is set to $\perp$ to mean that the corresponding node has no defined father. At any step of the computation, when a node v, not yet in the tree ($Stage(v) \neq WA$) receives a $<$**st_tok** $>$ message from its neighbor w, node v includes itself in the tree by changing its $Stage$ to WA. Moreover, in the same time, $Father(v)$ is set to w, set "Included"

Algorithm 15 Spanning tree procedure ($STP\_MPS(G, T; v_0; Stage, X, Y; Father; Sons)$)

- **Input :** A graph $G = (V, E)$ and v_0 the chosen root.
  - **Variables:**
    - $B(v)$: the set of (locally) ordered immediate neighbors of v which is an initial data,
    - $Stage(v)$: the state of v can take many values. The only used in this procedure is: "X" as the initial value, "WA" to mean that v has been included in the tree and "Y" to mean that v finished locally its computation;
    - $Father(v)$: is the father of v in the on-building spanning tree;
    - $Sons(v)$: an ordered list of the sons of v in the previous tree;
    - $Included(v)$: the set of v 's neighbors included for the first time in the on-building spanning tree;
    - $Terminated(v)$: the set of v 's neighbors that finished locally the computation of the spanning tree;
    - i, p, q : integer;
  - **Initialization:**
    - $\forall v \in V,\ Stage(v) = X$.
- **Results:** A spanning tree $T = (V, E_T)$ of G with root v_0 such that $\forall v \in V,\ Stage(v) = Y$.
- **Actions:**

$\mathbf{S}TA1$: {For the initiator v_0 only, execute once:}
$Stage(v_0) := WA$;
$Father(v_0) := \perp$;
$Sons(v_0) := \emptyset$;
$Included(v_0) := \emptyset$;
$Terminated(v_0) := \emptyset$;
for $i := 1$ **to** $deg(v_0)$ do send<**st_tok**> via port i;

$\mathbf{S}TA2$: {A message <**st_tok**> has arrived at v from port q}
1: **if** ($Stage(v) \neq WA$)
$Stage(v) := WA$;
$Father(v) := q$;
$Sons(v) := \emptyset$;
$Included(v) := \{q\}$;
$Terminated(v) := \emptyset$;
send<**st_son**> via port q;
2: **if** ($deg(v) = 1$)
$Stage(v) := Y$;
send<**st_back**> via port q;
3: **else**
for $i := 1$ **to** $deg(v)$ do
4: **if** ($i \neq q$) send<**st_tok**> via port i;
5: **else**
$Included(v) := Included(v) \cup \{q\}$;
6: **if** ($Included(v) = B(v)$ and $Sons(v) = \emptyset$) send<**st_back**> via port $Father(v)$;

$\mathbf{S}TA3$: {A message <**st_son**> has arrived at v from port q}
$Sons(v) := Sons(v) \cup \{q\}$;
$Included(v) := Included(v) \cup \{q\}$;

$\mathbf{S}TA4$: {A message <**st_back**> has arrived at v from port q}
$Terminated(v) := Terminated(v) \cup \{q\}$;
1: **if** ($Included(v) = B(v)$ and $Terminated(v) = Sons(v)$)
$Stage(v) := Y$
2: **if** ($Father(v) \neq \perp$)
send<**st_back**> via port $Father$;

is now composed of w and set "Terminated" is initialized to $\emptyset$. Finally, v informs w to add it in its set of sons sending a <**st_son**> message. At the reception of such a message, w adds v in both its sets of sons and in the set of nodes that are included in the tree ("Included"). So at each execution of the rule $STA2(1)$, the number of nodes not yet in the tree decreases by 1. When v finds all its neighbors already included in the tree, it applies either the action $STA2(2)$, $STA2(6)$ or $STA4(1)$. So it means that v has locally terminated its computation, then it informs its father $STA4(2)$.

The computation finishes when all the nodes v are such that $Stage(v) = Y$. And obviously we have a spanning tree of G rooted at v_0 defined by the third components (or the fourth components) of the variables of the nodes. The root of the spanning tree is then the unique node with its father equal to $\perp$.

First, we claim the following fact:

Fact 6.2.6

1. *The actions* $STA1$ *and* $STA2(4)$ *use* $(2\#E - (\#V - 1))$ <**st_tok**> *messages,*
2. *The actions* $STA2(6)$ *and* $STA4(2)$ *use* $(\#V - 1)$ <**st_back**> *messages,*
3. *The action* $STA2(2)$ *uses* $(V - 1)$ <**st_son**> *messages.*

Then, the following property holds:

Property 6.2.7 *Given a graph* $G = (V, E)$ *and a chosen node* v_0*. The* STP_MPS *builds a spanning tree of* G *rooted at* v_0 *using* $(2\#E + \#V - 1)$ *messages.*

6.3 The 2-Vertex Connectivity Test Protocol

In this section we present a protocol to test whether a given graph G is 2-connected. We will use an immediate application of the Menger theorem [Wes01] :

Proposition 6.3.1 *Let* T *be a spanning tree of a graph* G*.*

Then the graph G *is 2-connected iff* $\forall v \in V_G, \exists\, \overline{E} \subseteq E_G \setminus E_T$ *such that* $(V_{G\setminus v}, E_{T\setminus v} \cup \overline{E})$ *is a spanning tree of* $(V_{G\setminus v}, E_{G\setminus v})$ *with* $\#\overline{E} \leq \#SON(v)$*.*

Our protocol uses the following definition:

Definitions 6.3.2 *Let* $G = (V, E)$ *be a connected graph and* $T(Father, Sons)$ *be a spanning tree of* G *rooted at* v_0*. For each pair of nodes* u, v *in* G*, we say that* u *is reached by* v *in* G *iff there is a path in* G *linking* v *to* u*. Thus, we say that* v *is a succorer son of* v_d *iff the following holds:*

1. $v \in Sons(v_d)$,
2. *if* $v \neq v_0$ *then* v *is not reached by* v_0 *in* $G \setminus v_d$,
3. $\neg\exists\, u \in Sons(v_d)$ *such that* $u < v$ *and* v *is reached by* u *in* $G \setminus v_d$.

Then, the label $Suc(v)[v_d]$ *is set to* $true$*. Otherwise it is set to* $false$*.*

From this definition, we propose and prove the following theorem which is the base of our protocol:

Theorem 6.3.3 *Let $G = (V, E)$ be a connected graph. Then G is 2-connected iff $\forall\, T$ a spanning tree of G rooted at some node v_0 then only v_0 admits a succorer son.*

Proof. The correctness proof is by induction of the number of nodes in G. Suppose that $G = (V, E)$ is connected and $\#V = 2$. Let $V = \{u, v\}$ and let T be a spanning tree of G rooted at u. So, the property is trivially verified. Now, we suppose that $\#V = 3$. The only possible 2-connected graph composed of three nodes is its corresponding complete graph. Let $G_3 = (V_3, E_3)$ be such a graph and Let $V_3 = \{u, v, w\}$. We suppose that T is a spanning tree of G rooted at u. Let v be a succorer son of u. Since G_3 is complete, v reaches w in $G_3 \setminus u$, u reaches v (resp. w) in $G_3 \setminus w$ (resp. in $G_3 \setminus v$). So only u admits a succorer son. It is clear that for the opposite case, for example when v admits w as its succorer son, this means that w is not reached by u in $G_3 \setminus v$ and then G_3 is not 2-connected.

Now suppose that the property is verified for all connected graphs with $n - 1$ nodes. Consider a graph $G_n = (V_n, E_n)$ where $\#V_n = n$. We denote by $G_{n-1} = (V_{n-1}, E_{n-1})$ a sub-graph of G_n where $\#V_{n-1} = n - 1$. In other words, there is some node v_n such that $G_{n-1} = G_n \setminus v_n$.

Let T_{n-1} be a spanning tree of G_{n-1} rooted at u and let T_n be a spanning tree of G_n rooted at u. We have two cases: G_{n-1} is 2-connected or not.

For the first case, this means that all the sons of each node v in T_{n-1} are reached by the root u in $G_{n-1} \setminus v$. This is equivalent to say that all the sons of each node v in $T_n \setminus v_n$ are reached by the root u in $G \setminus \{v, v_n\}$. That is, if the sons of v_n in T_n are reached by u in $G_n \setminus v_n$ then G_n is 2-connected, otherwise it's not 2-connected.

To deal with the second case, we denote by $w \neq u$ a node which admits a succorer son in T_{n-1}. It suffices to check if such a node admits a succorer son in T_n. In G_{n-1}, there is some son w' of w in T_{n-1} not reached by u. This means that there is only one path linking such a son to the root in G_{n-1} and it is over w. In G, we have two possibilities: Node v_n is reached by the root in $G \setminus w$ and w' is also reached by v_n in $G \setminus w$ then, such a son will be necessary reached by the root. It means that there exists two paths linking such a son to the root: One over w and the other over u. Otherwise, the son of w remains not reached by the root in $G \setminus w$. This means that G is not 2-connected. □

6.3.1 Our Protocol

We now present an overview of our distributed test protocol. Along this description, we will give the main keys to understand why our protocol is based on a general and a constructive approach. It consists of the following phases: (1) the computation of the spanning tree called *Investigation tree*, denoted by Inv_T, of G with root v_0, (2) exploration of Inv_T to compute the succorer sons of each node of G.

In phase one, we use an algorithm as described in the previous section. This procedure constructs a spanning tree Inv_T of a given graph G rooted at v_0 with local detection of the global termination. It means that v_0 detects the end of the spanning tree computation of

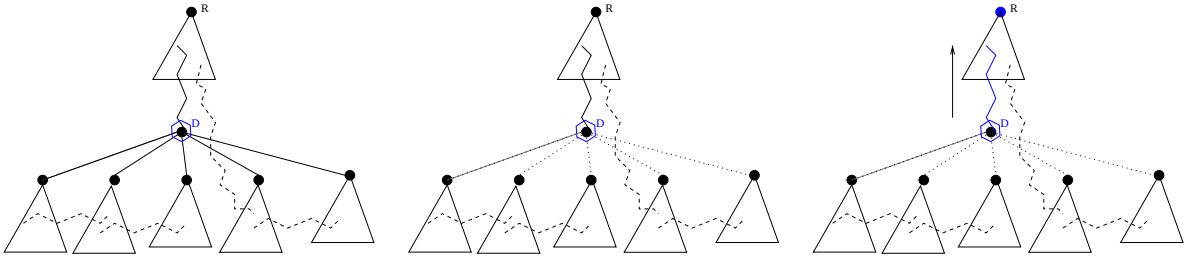


Figure 25: The 2-vertex connectivity test protocol (a,b,c)

G. In phase two, the tree Inv_T is explored using "depth-first trip" [Tel00]. When the trip reaches a node v_d, v_d does the following:

1. *disconnects.* Node v_d disconnects itself as shown in Figure 25.b.
2. *disconnected node v_d is the root.*
   (a) *configures.* An auxiliary spanning tree T of G, where v_0 is disconnected, is built.
   (b) *computes the first succorer son.* Node v_0 chooses its first son as its first succorer son. Let r be such a son. It will become the chosen root.
   (c) *maximal tree construction.* Node r is in charge to build a spanning tree T' of $G \setminus v_d$ rooted at itself.
3. *disconnected node v_d is not the root.*
   (a) *propagates.* Node v_d informs the root v_0 of the Inv_T about its disconnection (see Figure 25.c).
   (b) *configures.* An auxiliary tree T of G, where v_d is disconnected, is built.
   (c) *maximal tree construction.* The root of T starts the computation of its maximal tree T' of $G \setminus v_d$ as depicted in Figure 26.d.
   (d) *responds.* Eventually, v_0 constructs its maximal tree, so it responds to v_d (see Figure 26.e).
4. *succorer sons computation.* As shown in Figure 26.f, node v_d looks in its sons if there is some node not included in the maximal tree.
   i. If there exists such a son, and $v_d = v_0$: Node v_0 stops the treatment and states that G is not 2-connected.
   ii. If there exists such a son, and $v_d \neq v_0$: Node v_d stops the exploration informing the root v_0 which states that G is not 2-connected.
   iii. Otherwise, node v_d continues the exploration on Inv_T.

The algorithm terminates according to the following cases: (i) if the root admits more than one succorer son, (ii) if some node admits a succorer son, (iii) after the exploration of all nodes of G without the two previous cases. In the last case, there is only one succorer son which is the first son of v_0 the root of the chosen spanning tree Inv_T. Then, v_0 states that G is 2-connected. As we shall see, the aim of the configuration step represented above as *configures* is to prepare the application of the succorer sons computation of some node v_d. So, this phase

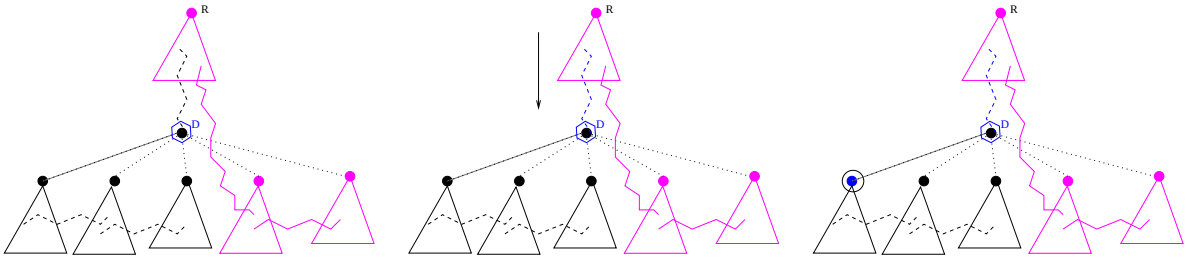


Figure 26: The 2-vertex connectivity test protocol (d,e,f)

initializes the set of labels erasing the trace of the last succorer sons computation. This phase is achieved by the root v_0 which updates an auxiliary tree T of G adding the simulation of the disconnection of the current explored node v_d [1].

The *maximal tree construction procedure* is applied on the node r which is in charge to build and to mark its maximal tree of $G \setminus v_d$. It works as follows: Let T be a spanning tree of G. We denote by T' the "on-building" tree of $G \setminus v_d$. This procedure is composed of the following steps:

1. *computes the base of the tree.* This base of T' is the sub-tree of $T \setminus v_d$ rooted at r. It is identified using a marking procedure (see below).
2. *search extension.* Using a "depth-first trip" [Tel00] on T' each node v is visited until one of the neighbor of v is not yet included in the current tree T'.
3. *connection is found.* The visited node v finds a neighbor u that is not in the "on-building" tree. Such a node is called a *connection entry* and v is named a *connection node*. Therefore, the edge (u, v) is added to T'. The sub-tree of T containing u is reorganized in such a way that u becomes its root using the *root changing procedure* (see below). Then, this sub-tree is included in T' and its nodes are marked as nodes of T'. Now, the search of isolated sub-trees is restarted on the extended T'.

Finally, r detects that it had extended as much as possible its tree and the procedure is terminated.

6.3.2 Example of Running

For instance, we consider a graph G with a spanning tree T with root v_d as shown in the first part of Figure 27. The numbers and names are only used to help the comments. So, the node v_d admits the node r, "gray" node, as its succorer son and the "white" nodes (1, 2, 3, 4) as other sons. Now we suppose that node r starts the execution of the *maximal tree construction procedure* in order to build a spanning tree of $G \setminus v_d$ rooted at r. As shown in the second part of Figure 27, T_r includes both sub-trees T_1 and T_2 because there is a path linking r, 1 and 2 without v_d. Whereas the sub-trees T_3 and T_4 are not included in T_r since the only path linking r, 3 and 4 contains v_d. Figure 28 shows an example of a run when v_d is not a cut-node. That's, after its deletion r succeeds to build a spanning tree of $G \setminus v_d$. All the sub-trees (1, 2, 3, 4) are included in T_r.

[1] Sons of v_d become "orphan nodes" or without "father".

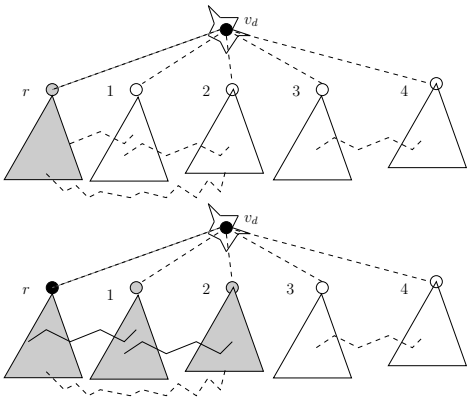


Figure 27: Example of the *maximal tree construction procedure* running when v_d is a cut-node

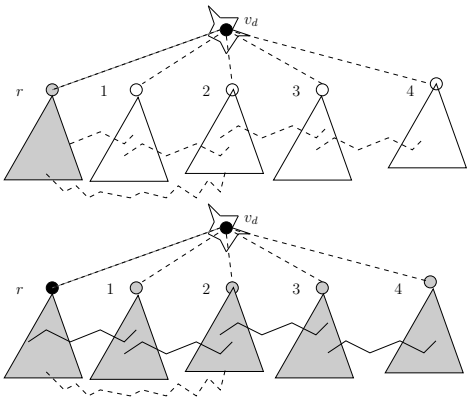


Figure 28: Example of the *maximal tree construction procedure* running when v_d is not a cut-node

In the sequel we propose an encoding of the protocol in the local computations model and an implementation in the message passing model. For each of them, we give the proofs of the correctness and a complexity analysis.

6.4 Encoding of our Protocol

In this section we propose an encoding of our protocol using graph relabeling systems as described in Section 1.2. So, the needed procedures are encoded in the same model and the analysis is done according to the given assumptions of such a model. We start by describing its main procedure: The *maximal tree construction procedure*. This procedure is composed of *five* rules and uses *two* procedures: The *marking procedure* and the *root changing procedure*. In the sequel we present a short description of these two procedures. The *maximal tree construction procedure* is more detailed.

Marking Procedure ($MP\_GRS(T; v_0; Stage, X, Y)$**)** This procedure is based on a broadcast algorithm on a tree T [Tel00]. The goal of this procedure is to transform the labels of each node v from $Stage(v) = X$ into $Stage(v) = Y$. More precisely, assuming that each node is labeled X in T. The node v_0, which is the root of T, starts the *marking procedure* relabeling itself $Stage(v_0) = W$. If v is labeled $Stage(v) = W$ and finds one of its sons w labeled X, then w sets its label W. When v is labeled $Stage(v) = W$ and it has no son or all its sons are marked, it marks itself (label "Y") and informs its father. Then, the father of such a node adds this son to its list of marked sons. The end of this procedure is detected by the root when all its sons are marked.

Property 6.4.1 *Given a tree $T = (V, E)$ rooted at v_0. Starting at v_0, the MP_GRS changes the labels of all the nodes in T from X into Y applying $2\#V$ steps.*

Root Changing Procedure ($RCP\_GRS(T; r; Treated)$**)** If a node r requests to replace the root v_0 of T, the nodes "between" [2] v_0 and r have to reorganize their lists of sons and their fathers. The node r starts the procedure relabeling itself W. If the node v finds its father w in the "old tree" [3] labeled A, it changes the label of w from A to W. This process will be applied for all the nodes in the "to be modified path". When v_0 is attained from w, its son in the "old tree", such that w is also in the "to be modified path", it will be relabeled A and sets its father to w. Then, the process is applied reversibly to all the nodes in the "to be modified path". Finally, node r detects the end of the last process when its "old" father has been treated.

Property 6.4.2 *Let $T = (V, E)$ be a tree rooted at v_0 and let r be a node in T. The RCP_GRS changes the original root of T with r applying, in worst case, $2\#V$ steps.*

Maximal Tree Construction Procedure ($MTCP\_GRS(G, T, T'; v_d, r)$**)** Given a node v_d of a spanning tree T of a graph G, this procedure builds a maximal tree T' of $G \setminus v_d$ rooted at r. We propose an encoding of this procedure using means of local computations, then we present an example of its run and its analysis. As we shall see, the algorithm is based on "depth first-trip"

[2] The nodes on the simple path linking v_0 and v.
[3] The sub-tree rooted at v_0

and uses the *marking procedure* to encode phase 1. The needed reorganization of phase 3 uses both *root changing procedure* and *marking procedure*.

Algorithm 16 Maximal tree construction procedure ($MTCP\_GRS(G, T, T'; v_d, r)$)

- **Input:** A graph $G = (V, E)$ with a spanning tree $T = (V, E_T)$, and a chosen node r.
  - **Labels:**
    - $Stage(v) \in \{A,\ D,\ Ended,\ F,\ SC,\ W\}$, $Father(v)$, $Sons(v)$, $B(v)$,
    - $To\_Explore(v)$, $Potential(v)$, $To\_Ignore(v)$.
  - **Initialization:**
    - $\forall\, v \in V \setminus \{v_d\}$, $Stage(v) = A$,
    - $\forall\, v \in Sons(v_d)$, $Father(v) = \perp$,
    - $\forall\, v \in V$, $To\_Explore(v) = Sons(v)$,
    - $\forall\, v \in V$, $Potential(v) = \emptyset$,
    - $\forall\, v \in V$, $To\_Ignore(v) = \emptyset$.
- **Result:** A maximal tree T' of $G \setminus v_d$ with root r. In this case, r is labeled $Ended$.
- **Rules:**

$MTR1$: **Node r labels the nodes of its sub-tree to F and starts the attempt of reconnection**

Precondition :
- $Stage(r) = A$

Relabeling :
- $Potential(r) := B(r) \setminus (Sons(r) \cup To\_Ignore(r))$
- $\mathbf{MP\_GRS(T_r; r; Stage, A, F)}$
- $Stage(r) := SC$

$MTR2$: **Node v is a connection node**

Precondition :
- $Stage(v) = SC$
- $u \in Potential(v)$
- $Stage(u) = A$

Relabeling :
- $\mathbf{RCP\_GRS(T(u) \setminus v_d; u)}$
- $\mathbf{MP\_GRS(T_u; u; Stage, A, F)}$
- $Father(u) := v$
- $Sons(v) := Sons(v) \cup \{u\}$
- $To\_Explore(v) := To\_Explore(v) \cup \{u\}$

$MTR3$: **Node u is labeled F**

Precondition :
- $Stage(u) = F$
- $u \in B(v)$ and $u \in Potential(v)$

Relabeling :
- $Potential(v) := Potential(v) \setminus \{u\}$
- $To\_Ignore(u) := To\_Ignore(u) \cup \{v\}$

$MTR4$: **Node v is not a connection node, it delegates the reconnection search to one of its sons u**

Precondition :
- $Stage(v) = SC$
- $Potential(v) = \emptyset$
- $u \in To\_Explore(v)$

Relabeling :
- $To\_Explore(v) := To\_Explore(v) \setminus \{u\}$
- $Potential(u) := B(u) \setminus (Son(u) \cup \{v\} \cup To\_Ignore(u))$
- $Stage(u) := SC$
- $Stage(v) := W$

$MTR5$: **The sub-tree with root v does not contain connection node**

Precondition :
- $Stage(v) = SC$
- $Stage(Father(v)) = W$ /* This condition is not necessary, but clarifies the rule. */
- $Potential(v) = \emptyset$
- $To\_Explore(v) = \emptyset$

Relabeling :
- $Stage(v) := Ended$
- **if** $(Father(v) \neq \perp)$ $Stage(Father(v)) := SC$

After the initialization, $\forall v \in V \setminus v_d$, $Stage(v) = A$. The chosen node r initiates the computation (see Figure 29.b): It is the only one that executes the rule $MTR1$. So r executes the *marking procedure* as shown in Figure 29.c. When r finishes the execution of this procedure, its sub-tree T_r in T is labeled F. It means that this sub-tree is included in the on-building tree. Node r switches to the "searching connection" phase ($Stage(r) = SC$, see Figure 30.d). Then, step by step, a tree T_r rooted at r is extended. At any time there is only one node labeled SC and this node is in charge to extend the tree it belongs to.

If a node v labeled SC finds a neighbor u labeled A (a connection entry, see Figure 30.e) in its "Potential" set, it executes the rule $MTR2$ to extend T_r : The sub-tree containing u is reorganized in such a way that u becomes its root and then this sub-tree is added to T_r using

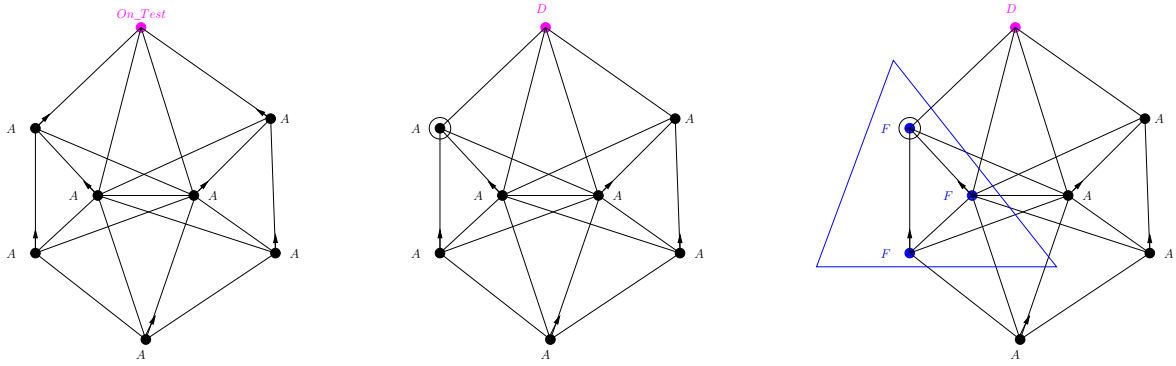


Figure 29: Maximal tree construction procedure (a,b,c)

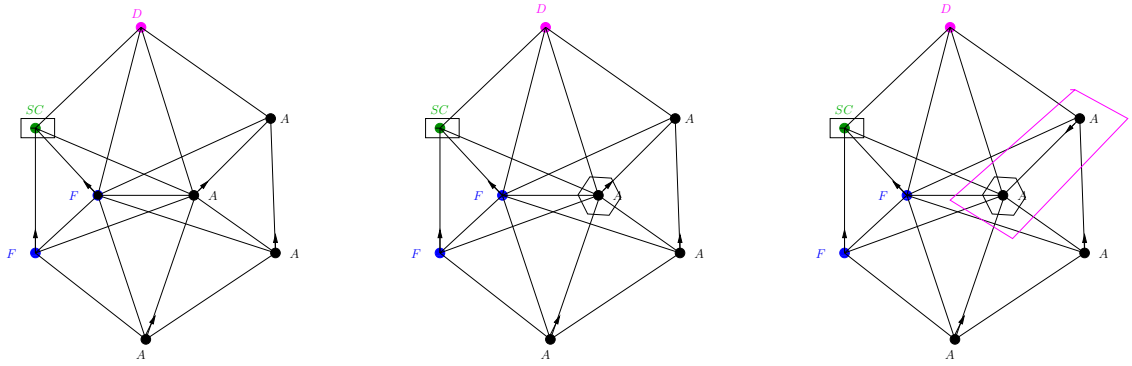


Figure 30: Maximal tree construction procedure (d,e,f)

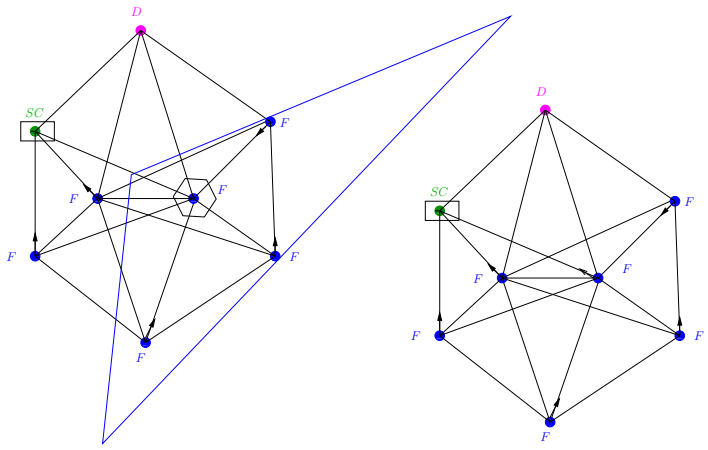


Figure 31: Maximal tree construction procedure (g,h)

the edge (v, u). Let $T(u) \setminus v_d$ be the tree that contains the connection entry u. Node u starts the execution of the *root changing procedure* as shown in Figure 30.f. The *marking procedure* is then applied to the tree rooted at u in order to relabel its nodes to F (see Figure 31.g). Now, u is added to T_r, label $Father$ of u is set to v and u is added in both $Sons(v)$ and $To\_Explore(v)$ (see Figure 31.h).

In parallel, the rule $MTR3$ is used by a node w to update the lists "Potential" and to prepare their computations using the lists "To_Ignore". If a node w is labeled F, then it cannot be a candidate for reconnection. Since it is yet in the tree, it must be avoided from the lists "Potential" of its neighbors. Moreover, the neighbors of w yet in the tree (they are labeled F, W , or SC) have to be ignored in its "Potential". When a node v labeled SC has an empty list "Potential" (this happens when it has no neighbor labeled A and when all it neighbors labeled F have avoided themselves from its list "Potential"), v executes the rule $MTR4$ to indicate that it does not succeed to find an extension for T_r. So it transfers the label SC to one of its sons that restarts the process of reconnection.

Eventually, node v in T_r has extended as much as possible T_r. Then, it informs with rule $MTR5$ its father ($Stage(v) = Ended$). Now, this last node can proceed with the extension search from one of its other sons. When $Stage(r) = Ended$, one of the largest tree rooted at r of $G \setminus v_d$ has been computed. Thus, r detects the end of the *maximal tree construction procedure*.

Now we propose a lemma which shows the tasks invoked by the execution of the *maximal tree construction procedure* when applied to a graph G about one of its node v_d.

Lemma 6.4.3 *After the execution of $MTCP\_GRS$ on a graph $G = (V, E)$, node r constructs a maximal tree including all the nodes reached by r in $G \setminus v_d$.*

Proof. To prove this lemma we proceed by contradiction. Assume that after the execution of the *maximal tree construction procedure* there is some node u, a son of v_d, reached by the root r in $G \setminus v_d$ with label A. It means that it is not included in the constructed tree. Now we show that such a behavior happens when there is no path linking u to r in $G \setminus v_d$ which contradicts the fact that u is reached by r in $G \setminus v_d$. At the start of the execution of this procedure all the nodes of $G \setminus v_d$ are labeled A. Since our algorithm is based on a "depth-first" exploration of the base of the tree and then on a "depth-first" exploration of the building tree, all the nodes linked to the root are explored (rules $MTR4$, $MTR5$, $MTR2$) and labeled $Ended$. Thus, at the end of the execution of this procedure, a node which remains with label A is the only one not linked to the root and then not reached by the root. □

For the time complexity, the worst case running time of the *maximal tree construction procedure* corresponds to the following:

1. The rule $MTR1$ can only be applied once. So its cost is due to the application of the *marking procedure* to the nodes of the sub-tree rooted at r : At most $2\#V$ rules.
2. The rule $MTR2$ can only be also applied once for each sub-tree rooted at some son of v_d different from r. So its cost is due to the application of the *marking procedure* to each sub-tree rooted at the sons of v_d except r augmented by the cost of the *root changing procedure*. Then, the cost of both rules $MTR1$ and $MTR2$ is $4(\#V - 1 - size(r)) + 2\, size(r)$, where $size(r)$ is the size of the sub-tree on which the rule $MTR1$ is applied. So the cost of $MTR1$ and $MTR2$ is bounded by $4\#V$.

3. The rule $MTR3$ is applied once by each node to update its "To_Ignore" and the "Potential" of its neighbors not in the tree. So, a priory, the cost of the application of this rule is $2\#E$, but since we use the list "To_Ignore" only $\#E$, applications are necessary. Moreover, the edges of the tree have not to be taken into account and also the sons of v_d. Finally, the cost of the use of $MTR3$ is $\#E - (\#V - 2)$.

4. Each of the rules $MTR4$ and $MTR5$ is applied, in the worst case, $\#V - 2$ times to encode the "depth-first trip" mechanism.

Then,

Lemma 6.4.4 *The $MTCP\_GRS$ requires at most the application of $\#E + 4\#V$ rules.*

6.4.1 2-Vertex Connectivity Test Algorithm

In this part we present an encoding of our protocol referred to in the following as $2\mathcal{VCT}\_\mathcal{GRS}$ algorithm. This algorithm uses *nine* rules and *three* procedures: The *spanning tree construction procedure* to construct the investigation tree, the *simple path propagator procedure* to encode phase $3(a)$ and its reversible version to encode phase $3(d)$ and the *configuration procedure* to encode both phases $2(a)$ and $3(b)$. Phases $2(c)$ and $3(c)$ are achieved using the *maximal tree construction procedure* presented above. Now, we present a short description of the two new procedures. Then, we give a full description of the algorithm and an example of its run.

Simple Path Propagator Procedure ($SPPP\_GRS(T; v; Traversed, X, Y)$) The aim of this procedure is to propagate an information along a simple path linking some node v to the root of the tree $T = (V, E_T)$. We will introduce the label $Traversed$ to notice the expected distinguished simple path. Initially, all the nodes u are labeled $Traversed(u) = X$. At the end of the application of this procedure, each node u in the simple path is labeled $Traversed(u) = Y$. To encode this procedure one can use the first part of the *root changing procedure* augmented by the update of the label $Traversed$.

Property 6.4.5 *Given a tree $T = (V, E)$ rooted at v_0. Let v be a node in T. By executing the $SPPP\_GRS$, if v sends a message to v_0, then this message will be received by v_0 applying, in the worst case, $\#V$ steps.*

Configuration Procedure ($CP\_GRS(Inv\_T, T; v_0, v_d; Stage, X, Y; Required)$) This procedure allows to do the required initializations of the *maximal tree construction procedure* in order to compute the succorer sons. So this procedure is executed by the root before to start the application of the $MTCP\_GRS$ about v_d removing the trace of the last computations. Node v_0, which is, the root of the tree T executes a cycle involving all the nodes of the graph to initialize the labels used in the $MTCP\_GRS$. The initialization cycle may be encoded using an extension of the *marking procedure* without adding extra operations. Moreover, v_d prepares its disconnection, so each of its sons sets its label "Father" to $\perp$.

Property 6.4.6 *Let $G = (V, E)$ be a graph. Let Inv_T be a spanning tree of G rooted at v_0 and let v_d be a node in G. Starting at v_0, the CP_GRS configures T as a spanning tree of G using Inv_T and disconnects v_d. The cost of this module is only the one of the MP_GRS which is $O(\#V)$ steps.*

Now, we give the required initializations to encode our distributed protocol to test the 2-vertex connectivity of a graph. Such initializations concern the following labels:

$Father(v), Sons(v), Potential(v), To\_Ignore(v), To\_Explore(v), Treated(v), Traversed(v)$. So, we denote by $Required\_CP$ these labels and their required initialization. In other words, the use of the term $Required\_CP$ in the relabeling part of the rewriting rule is equivalent to:

1. $Father(v) := InvFather(v)$
2. $Sons(v) := InvSons(v)$
3. $Potential(v) := \emptyset$
4. $To\_Ignore(v) := \emptyset$
5. $To\_Explore(v) := Sons(v)$
6. $Treated(v) := \emptyset$
7. $Traversed(v) := 0$

To encode efficiently our protocol, we propose an extension of the *root changing procedure* as follows. This procedure changes the root of the sub-trees after the disconnection of some node v_d. Alternatively, to collect the state of the sons of v_d after its test to compute the set of its succorer sons, we propose to do this in the *root changing procedure*. So we add the label $Treated$ as the set of sons of v_d included in the current maximal tree. When v the son of v_d is attained during the execution of this procedure, the following is done:

$Treated(InvFather(v)) := Treated(InvFather(v)) \cup \{v\}$.

That is, this procedure does not cost extra time because it is incorporated in the other procedures.

As depicted in Algorithm 17, let $G = (V, E)$ be a graph and $v_0 \in V$ be a node to launch and supervise the test algorithm. We have $\forall v \in V$, $InvStage(v) = N$. The node v_0 starts the computation: It builds a spanning tree Inv_T of G rooted at v_0 ($VCTR1$). Subsequently, all the vertices of G are such that $InvStage = A$.

Since the test scheme is based on the use of a "depth-first trip" exploration on the tree Inv_T, the root v_0 is the only one able to start the trip with the rule $VCTR2$. For the particular case of v_0, it is examined as follows: First, it chooses one of its son as its succorer son. This son is in charge to build a maximal tree of $G \setminus v_0$ after the configuration phase. For each other nodes, when it is ready to be examined it switches its label to D ($VCTR5$).

The examination phase of the node v_d consists on the test if for a given spanning tree T of G deprived of v_d, v_0 is able to build a maximal tree T' of $G \setminus v_d$ including the sons of v_d in T. It proceeds as follows: The node v_d informs v_0 about its attention to compute its succorer sons applying the $SPPP\_GRS$ ($VCTR5$). Then v_0 configures the labels using the CP_GRS which cleans the spanning tree T' of G rooted at v_0 and disconnecting v_d ($Stage(v_d) = D$, all the sons of v_d set their fathers to $\perp$). At the end of the *configuration procedure*, v_0 starts to build, if possible, its maximal tree T' of $G \setminus v_d$ using the $MTCP\_GRS$ presented above ($VCTR6$).

Eventually v_0 constructs such a tree and responds to v_d using a reversible version of the $SPPP\_GRS$. When v_d receives the information from v_0, it looks in the set "Treated" computed

Algorithm 17 2-Vertex connectivity test algorithm ($2\mathcal{VCT}\_\mathcal{GRS}$)

- **Input:** A graph $G = (V, E)$ and a node $v_0 \in V$.
- **Labels:**
  - $Stage(v)$, $B(v)$, $Treated(v)$,
  - $InvStage(v) \in \{A, D, KO, N, OK, Over, W\}$,
  - $Traversed(v) \in \{O, 1\}$,
  - $InvFather(v)$, $InvSons(v)$.
- **Initialization:**
  - $\forall v \in V$, $InvStage(v) = N$, $Treated(v) = \emptyset$ and $Traversed(v) = 0$,
  - $\forall\, v \in V, \forall u \in B(v)\ SUC(v)[u] = false$.
- **Result:** Two possible results :
  - $InvStage(v_0) = Over$ to mean that the graph G is 2-connected.
  - $InvStage(v_0) = KO$ to mean that the graph G is not 2-connected.
- **Rules:**

$VCTR1$: **Node v_0 starts to build the tree $Inv\_T(InvFather, InvSons)$**

Precondition :
- $InvStage(v_0) = N$

Relabeling :
- $\mathbf{STP\_GRS(G, Inv\_T; v_0; InvStage, N, A; InvFather; InvSons)}$ /*all the nodes v satisfy : $InvStage(v) = A$.*/

$VCTR2$: **Node v_0 initializes the test $Inv\_T(InvFather, InvSons)$ /*v_0 is the only node such that $InvFather(v_0) = \perp$ [4]*/**

Precondition :
- $InvStage(v_0) = A$
- $InvFather(v_0) = \perp$
- $r \in Sons(v_d)$

Relabeling :
- $InvStage(v_0) := D$
- $\mathbf{CP\_GRS(Inv\_T, T; v_0, v_0; Stage, N, A; Required\_CP)}$
- $\mathbf{Suc(r)[v_0] := true}$
- $\mathbf{MTCP\_GRS(G, T, T_r; v_0, r)}$

$VCTR3$: **Node v_d has not found a succorer son**

Precondition :
- $InvStage(v_d) = D$
- $Traversed(v_d) = 0$
- $Treated(v_d) = Sons(v_d)$

Relabeling :
- $InvStage(v_d) := OK$

$VCTR4$: **Node v_d finds a succorer son**

Precondition :
- $InvStage(v_d) = D$
- $Traversed(v_d) = 0$
- $v \in Sons(v_d)$
- $Stage(v) = A$

Relabeling :
- $InvStage(v_d) := KO$

$VCTR5$: **Node v ends the computation of its succorers sons, one of its son will become activated**

Precondition :
- $InvStage(v) = OK$
- $v_d \in InvSons(v)$

Relabeling :
- $InvSon(v) := InvSon(v) \setminus \{v_d\}$
- $InvStage(v) := W$
- $InvStage(v_d) := D$
- $Traversed(v_d) := 1$
- $\mathbf{SPPP\_GRS(Inv\_T; v_d; Traversed, 0, 1)}$

$VCTR6$: **Node v_0 is informed about the activation of some node v_d, it computes its maximal tree and then informs v_d.**

Precondition :
- $InvFather(v_0) = \perp$
- $Traversed = 1$

Relabeling :
- $\mathbf{CP\_GRS(Inv\_T, T; v_0, v_d; Stage, Ended, A; Required\_CP)}$
- $\mathbf{MTCP\_GRS(G, T, T_{v_0}; v_0)}$
- $\mathbf{SPPP\_GRS^{-}1(Inv\_T; v_0; Traversed, 1, 0)}$

$VCTR7$: **Node v ends the examination of its sub-tree**

Precondition :
- $InvStage(v) = OK$
- $InvSons(v) = \emptyset$
- $InvFather(v) \neq \perp$

Relabeling :
- $InvStage(v) := Over$
- $InvSons(InvFather(v)) := InvSons(InvFather(v)) \setminus \{v\}$
- $InvStage(InvFather(v)) := OK$

$VCTR8$: **Node v_0 detects the end and the success of the test algorithm**

Precondition :
- $InvStage(v) = OK$
- $InvSons(v) = \emptyset$
- $InvFather(v) = \perp$

Relabeling :
- $InvStage(v) := Over$

$VCTR9$: **The information about the failure of the test is transferred to the root v_0**

Precondition :
- $InvStage(v_d) = KO$
- $InvFather(v_d) \neq \perp$

Relabeling :
- $InvStage(v_d) := Over$
- $InvStage(Father(v_d)) := KO$

during the execution of the RCP_GRS induced by the execution of the $MTCP\_GRS$, if all its sons in T are included in T'. For the particular case of the root ($v_0 = v_d$), it detects itself the end of the work of its chosen succorer son. Then, it does the same work as the other nodes.

If v_d finds at least one son v labeled $Stage(v) = A$: The test fails ($VCTR4$) and then stops. The current node v_d will be labeled $InvStage(v_d) = KO$. There are two possibilities: If $v_d \neq v_0$ then it informs its father in Inv_T applying the rule $VCTR9$ until the root v_0. Second, if the test fails about the root v_0 ($VCTR4$), then v_0 stops the test procedure ($InvStage(v_d) = KO$). In the two cases, v_0 states that the graph G is not 2-connected.

In the other case("Treated=Sons"), v_d continues the exploration ($VCTR3$, $VCTR5$) choosing one of its not yet observed son. It transfers the label D to such a son.

Eventually a node v_d finishes the exploration of all the nodes in its sub-tree ($InvSons(v_d) = \emptyset$). So it informs its father that the computation is over ($VCTR7$). Now v_d is removed from the list $InvSons$ of its father which then can continue the exploration choosing one of its not yet observed son. Furthermore, only the root v_0 detects the end of the "trip" when all its sons are over. It means that all the nodes are examined and succeeded. So v_0 applies the rule $VCTR8$ to affirm that the graph G is 2-connected.

6.4.2 Overall Proofs and Complexities Analysis of the $2\mathcal{VCT}\_\mathcal{GRS}$

Now, we show the correctness of the $2\mathcal{VCT}\_\mathcal{GRS}$ algorithm using a scheme based on the procedures properties as explained in the section about the system model. Then, the analysis is closed with time and space complexity measures.

Theorem 6.4.7 *The $2\mathcal{VCT}\_\mathcal{GRS}$ algorithm presented above implements a distributed test of the 2-vertex connectivity of a graph.*

Proof. First, we show that all the nodes will be observed: Since G is connected, the investigation tree will necessary include all the nodes of G, and since the exploration follows a "depth-first trip" on the investigation tree all the nodes will be visited and observed. Second, the correctness proof is by induction on the number of nodes in G.

Suppose that $G = (V, E)$ is connected and $\#V = 2$. The algorithm terminates after testing G as follows. Let $V = \{u, v\}$ and let u be the root of the investigation tree. It applies the rule $VCTR1$ to build an investigation tree. Then, it starts the test ($VCTR2$). Node u is disconnected, then its first succorer son v builds a spanning tree of $G \setminus v$ applying the rule $VCTR2$. It's composed only of v. So u is labeled $InvStage(u) = OK$. Then v is examined ($VCTR5$). Then u applies the rule $VCTR6$ to construct its maximal tree and then informs v. Now v has no son, then it informs its father u about this. Finally, u applies rule $VCTR8$. Thus the algorithm operates correctly in this case.

Now, we suppose that $\#V = 3$. The only possible 2-connected graph composed of three nodes is its corresponding complete graph. Let G_3 be such a graph, such that $V_3 = \{u, v, w\}$. Let u be the root of its corresponding investigation tree. If the investigation tree has two leaves the test is restricted to u.

Since G_3 is complete, the succorer son of u reconnects the third node. During the observation of v(resp. w), u succeeds to reconnect w(resp. u). So u affirms that G_3 is 2-connected. For a connected but not complete graph of three nodes, one of them fails the test. So the root

affirms that such a graph is not 2-connected.

Now suppose that the algorithm works correctly on all connected graphs with $n-1$ nodes. Consider applying the algorithm on a connected graph $G = (V, E)$ where $\#V = n$. We denote by $G' = (V', E')$ the sub-graph of G composed of the first $(n-1)^{th}$ yet observed nodes and let u the last node to be tested. In other words, $G' = G \setminus u$. We have two cases:

First, there is some node w which has a son not reached by the root in $G' \setminus w$. This means that there is only one path linking such a son to the root in G' and it is over w. In G, we have two possibilities: u is reached by the root in $G \setminus w$ and a son of w is also reached by u in $G \setminus w$ then, such a son will be necessary reached by the root. It means that there exists two paths linking such a son to the root: One over w and the other over u. Otherwise, the son of w remains not reached by the root in $G \setminus w$.

Second, since we would like to prove the property for any graph and then for any investigation tree, the position of u in the investigation tree is not a priory fixed as a leaf for example. From the hypothesis, $G' = G \setminus u$ is 2-connected. We must show whether G is 2-connected. We have two possibilities: u is observed positively or not.

For the first one, this means that all the sons of u are reached by the root in $G \setminus u$. From the hypothesis, for each node $v \in G'$, the sons of v are reached by the root in $G' \setminus v$ and then reached by the root in $G \setminus v$. Thus, for each node $w \in G$, the sons of w are reached by the root in $G \setminus w$. Therefore, G is 2-connected.

For the second one, this means that there is some son of u not reached by the root in $G \setminus u$: So the algorithm stops and G is not 2-connected. □

For the time complexity, we will show an example of the running of the test algorithm when the *maximal tree construction procedure* succeeds for every node. That is the worst case.

1. The rule $VCTR1$ is applied once. Its cost is due to the application of the *spanning tree procedure* to the graph G.

2. The *configuration procedure* and the *maximal tree construction procedure* are applied for each node in the investigation tree to compute its succorer sons. For the root, the computation is done by the rules $VCTR2$. For the other nodes, the computation is augmented by the use of the *simple path propagator procedure* and its reversible version. So the computation is done using the rules $VCTR5$ and $VCTR6$. Then, the cost of the application of the rules $VCTR2$, $VCTR5$, $VCTR6$ is due to the application of such procedures.

3. The rule $VCTR3$ is applied by each node in the investigation tree which succeeds the test procedure.

4. The rules $VCTR7$, $VCTR8$ are used to implement the "depth-first trip" on the tree $Inv\_T = (V, E_T)$. So, the cost of such exploration is $2\#V - 1$ rules.

Without loss of generality we denote by V the set of the nodes and by E the set of edges of all the graphs structures used in all the procedures. From the previous, we can claim the following:

- $COST(STP\_GRS(G, T; v_0; Father; Sons; Stage, X, Y), time) = \#E + \#V + 1$ (see Property 6.2.5).

- $COST(MTCP\_GRS(G,T,T';v_d,r),time) = \#E + 4\#V$ (see Lemma 6.4.4).
- $COST(CP\_GRS(Inv\_T,T;v_0,v_d;Stage,X,Y;Required),time) = 2\#V$ (see Property 6.4.6).
- $COST(SPPP\_GRS(T;v;Traversed,X,Y),time) = \#V$ (see Property 6.4.5).
- $COST(SPPP^{-1}\_GRS(T;v;Traversed,X,Y),time) = \#V$ (see Property 6.4.5).

Then,

$COST(2\mathcal{VCT}\_\mathcal{GRS}(G),time) = (\#V+1) \times \#E + 8\#V^2 + 2\#V$. Therefore, the time requirement of the $2\mathcal{VCT}\_\mathcal{GRS}$ algorithm is in $O(deg(G) \times \#V^2)$.

Thus,

Lemma 6.4.8 *Given a graph $G = (V,E)$. The $2\mathcal{VCT}\_\mathcal{GRS}$ algorithm tests the 2-vertex connectivity of G in $O(deg(G) \times \#V^2)$ time.*

Lemma 6.4.9 *The space complexity requirement of the $2\mathcal{VCT}\_\mathcal{GRS}$ algorithm is in $O(deg(G) \times \log deg(G))$ bits per node* [5].

Proof. Each node v is labeled $L(v)$ using the two following components:

1. $B(v)$, $Included(v)$, $Terminated(v)$, $Sons(v)$, $Feedbacks(v)$, $To\_Explore(v)$, $Potential(v)$, To_Ignore, $Treated(v)$, $InvSons(v)$.
2. $Stage(v)$, $Father(v)$, $InvStage(v)$, $InvFather(v)$, $Traversed(v)$, $SUC(v)$

Thus, to encode the first component of a label, every node needs to maintain subsets of its neighbors as descendants, for the set of sons for example. So, every node v needs $10\,deg(v) \times \log deg(v)$ bits to store this component. By taking into account the other labels used in the second component of labels, we can claim that $COST(2\mathcal{VCT}\_\mathcal{GRS}(G),bits) = 10\,deg(G) \times \log deg(G) + 2\,\log 7 + (3 + deg(G))\,\log 2$. □

The following result completes the analysis.

Theorem 6.4.10 *The $2\mathcal{VCT}\_\mathcal{GRS}$ algorithm encodes a distributed computation of the 2-vertex connectivity test of a graph. When the $2\mathcal{VCT}\_\mathcal{GRS}$ is applied to a graph $G = (V,E)$, its time complexity is in $O(deg(G) \times \#V^2)$ and its space complexity is in $O(deg(G) \times \log deg(G))$ bits per node.*

6.5 Implementation of our Protocol

Now, we present an implementation of our protocol in the message passing model as given in Section 1.3. So, the used procedures are encoded in the same model and the analysis is achieved according to the given assumptions of such a model. We start by describing its main procedure: *maximal tree construction procedure*. This procedure is composed of *thirteen* actions and uses *two* procedures: The *marking procedure* and the *root changing procedure*. Now, we present a short description of these two procedures. Then, we give a description of the main procedure.

[5] We use $\log x$ to denote $\log_2 x$.

Marking Procedure ($MP\_MPS(T; v_0; Stage, X, Y)$) After the execution of the algorithm, the label of each node v has been changed into $Stage(v) = Y$. More precisely, assuming that each node has its variable "Stage" set to X in T. The node v_0 starts the procedure. It modifies its stage: $Stage(v_0) = W$. At any time, a node v with "Stage" equal to W broadcasts a <**m_tok**> message to all its sons in T. If v receives such a message, its stage will be set to W if it has sons. Otherwise, if it has no son or all its sons are marked: It marks itself Y and informs its father about this change sending a <**m_back**> message. Then, when the father of such a node receives such a message, it adds this son to its list of marked sons. The end of this procedure is detected by the root when it receives a <**m_back**> message from each of its sons in T.

Property 6.5.1 *Given a tree $T = (V, E)$ rooted at v_0. Starting at v_0, the MP_MPS marks T using $(\#V - 1)$ <**m_tok**> messages and $(\#V - 1)$ <**m_back**> messages.*

Root Changing Procedure ($RCP\_MPS(T; r)$) This procedure will be used to change the root of a tree T. So if a node r has to replace the root v_0, it starts the procedure: It sets its variable "Stage" to W and sends a <**rc_tok**> message to its father in the "old tree" [6]. At the reception of such a message by $v \neq v_0$, it changes its stage from A to W and sends a <**rc_tok**> message to its father in the "old tree". When v_0, is attained from w, its son in the "old tree", v_0 sets its stage to A and its father to w and sends a <**rc_son**> message to w. Then, the process is applied reversibly to all the nodes in the "to be modified path".

Property 6.5.2 *Given a tree $T = (V, E)$ rooted at v_0. Let r be a node in T. The RCP_MPS replaces the original root with r using at most $(2\#V - 2)$ messages.*

Maximal Tree Construction Procedure ($MTCP\_MPS(G, T, T'; v_d, r)$) Given a node v_d of a spanning tree T of a graph G, the *maximal tree construction procedure* builds, if possible, a spanning tree T' of $G \setminus v_d$ rooted at r. In the following, we propose an implementation of this procedure in the message passing model. For a sake of clarity, we also present an example of its run. Then, we propose two lemmas to exhibit its task and to analyze its cost. The algorithm is based on "depth first-trip" and uses both the *marking procedure* and the *root changing procedure* presented above.

As described in Algorithm 18, let $G = (V, E)$ be a graph and let $T = (V, E_T)$ be a spanning tree of G. We consider a node v_d as a disconnected node and r a chosen node (often the root) to build the maximal tree of $G \setminus v_d$. When we start the algorithm all the nodes v of V have their variables $Stage$ valued A except v_d which is valued D. The node r initiates the computation: It is the only one that executes the action $MTA1$. So, r executes the *marking procedure*. When r finishes the execution of this procedure, its sub-tree in T is marked F. It means that this sub-tree is included in the on-building tree T'. Node r switches to the "searching connection" phase ($Stage(r) = SC$). At any time there is only one node this phase and this node is in charge to extend the construction of the tree it belongs to.

Node v where $Stage(v) = SC$ and with no empty "Potential" set starts the search of connection sending a <**mt_sc**> message via one of its potential port (sub-action $MTA2(1)$). If a node u labeled A (a connection entry) receives such a message, it executes action $MTA3(1)$ to extend T'. So its sub-tree is added to T' after a reorganization.

[6] The sub-tree rooted at v_0

Algorithm 18 Maximal tree construction procedure ($MTCP\_MPS(G, T, T'; v_d, r)$)

- **Input:** A graph $G = (V, E)$ with a spanning tree $T = (V, E_T)$, and a chosen node r.
  - **Variables:**
    - $Stage(v) \in \{A,\ D,\ Ended,\ F,\ SC,\ W\}$, $Father(v)$, $Sons(v)$, $B(v)$,
    - $To\_Explore(v)$, $Potential(v)$, $To\_Ignore(v)$.
  - **Initialization:**
    - $\forall\, v \in V \setminus \{v_d\}$, $Stage(v) = A$
    - $\forall\, v \in Sons(v_d)$, $Father(v) = \perp$,
    - $\forall\, v \in V$, $To\_Explore(v) = Sons(v)$,
    - $\forall\, v \in V$, $Potential(v) = \emptyset$,
    - $\forall\, v \in V$, $To\_Ignore(v) = \emptyset$.
- **Result:** A maximal tree T' of $G \setminus v_d$ with root r. In this case, r is labeled $Ended$.
- **Actions:**

$\mathbf{M}TA1$: {For the initiator r only, execute once:}
$Potential(r) := B(r) \setminus Sons(r)$;
$\mathbf{MP\_MPS}(\mathbf{T_r}; \mathbf{r}; \mathbf{Stage}, \mathbf{A}, \mathbf{F})$;
$Stage(r) := SC$;

$\mathbf{M}TA2$: {For each node v such that $Stage(v) = SC$, execute once:}
1: **if** ($Potential(v) \neq \emptyset$)
$Stage(v) := W$;
Let p be a port number in $Potential(v)$;
$Potential(v) := Potential(v) \setminus \{p\}$;
send<**mt_sc**> via port p;
else
2: **if** ($To\_Explore(v) \neq \emptyset$)
$Stage(v) := W$;
Let p be a port number in $To\_Explore(v)$;
$To\_Explore(v) := To\_Explore(v) \setminus \{p\}$;
send<**mt_deleg**> via port p;
3: **else**
$Stage(v) := Ended$;
send<**mt_ended**> via port $Father(v)$;

$\mathbf{M}TA3$: {A message <**mt_sc**> has arrived at v from port q}
1: **if** ($Stage(v) = A$)
$Stage(v) := CE$;
$\mathbf{RCP\_MPS}(\mathbf{T(v)} \setminus \mathbf{v_d}; \mathbf{v})$;
$\mathbf{MP\_MPS}(\mathbf{T_v}; \mathbf{v}; \mathbf{Stage}, \mathbf{A}, \mathbf{F})$;
$Father(v) := q$;
send<**mt_son**> via port q;
2: **else**
$To\_Ignore(v) := To\_Ignore(v) \setminus \{q\}$;
send<**mt_fail**> via port q;

$\mathbf{M}TA4$: {A message <**mt_son**> has arrived at v from port q}
$Stage(v) := SC$;
$Sons(v) := Sons(v) \cup \{q\}$;
$To\_Explore(v) := To\_Explore(v) \cup \{q\}$;

$\mathbf{M}TA5$: {For each node v such that $Stage(v) = F$ or $Stage(v) = D$, execute once:}
for $i := 1$ **to** $deg(v)$ do
if ($i \notin To\_Ignore(v)$) send<**mt_included**> via port i;

$\mathbf{M}TA6$: {A message <**mt_included**> has arrived at v from port q }
$To\_Ignore(v) := To\_Ignore(v) \cup \{p\}$;

$\mathbf{M}TA7$: {A message <**mt_fail**> has arrived at v from port q}
$Stage(v) := SC$;

$\mathbf{M}TA8$: {A message <**mt_deleg**> has arrived at v from port q}
$Potential(v) := Potential(v) \setminus (Sons(v) \cup To\_Ignore(v))$;
$Stage(v) := SC$;

$\mathbf{M}TA9$: {A message <**mt_ended**> has arrived at v from port q}
1: **if** ($To\_Explore(v) \neq \emptyset$)
Let p be a port number in $To\_Explore(v)$;
$To\_Explore(v) := To\_Explore(v) \setminus \{p\}$;
send<**mt_deleg**> via port p;
2: **else**
$Stage(v) := Ended$;
3: **if** ($Father(v) \neq \perp$)
send<**mt_ended**> via port $Father(v)$;

Let $T(u) \setminus v_d$ be a tree that contains the connection entry u. Node u starts the execution of the *root changing procedure*. The *marking procedure* is then applied to the tree rooted at u. Now, u is ready to include the tree T', labels $Father$ of u is set to v and a <**mt_son**> message is sent to v. When a <**mt_son**> message has arrived at v from u, node u is added in both $Sons(v)$ and $To\_Explored(v)$ (action $MTA4$). It is now ready to start the search of another connection entry.

At any time, actions $MTA5$ and $MTA6$ are used to update the lists "Potential" and to prepare their computations using the lists "To_Ignore". These actions may be executed concurrently with the other actions. If a node v is labeled F or D then it cannot be a candidate for reconnection since, either it is yet in the tree and so it must be avoided from the lists "Potential" of its neighbors or it is the node v_d. Moreover, the neighbors of v yet in the tree (they are labeled F, W, SC) have to be ignored in its "Potential".

When a node v labeled SC has an empty list "Potential", which arises when it has no neighbor labeled A and all its neighbors are labeled F have avoided themselves from its list "Potential", v executes the actions $MAT2(2)$ to mean that it does not succeed to find an extension for T'. So it transfers the label SC to one of its sons sending the <**mt_deleg**> message. The list of the candidate nodes to a reconnection from the "chosen" son is computed ("Potential") executing action $MTA8$ followed by action $MTA2$.

Eventually, some node v in T' has extended its sub-tree as much as possible ($Stage(v) = Ended$), it informs, with action $MTA2(3)$ (or $MTA9(3)$), its father sending the <**mt_ended**> message. The father which receives such a message can proceed to the extension search from one of its other sons executing action $MTA9$.

Furthermore, only r detects the end of the extension search ($Father(r) = \perp$). When $Stage(r) = Ended$ one of the largest tree rooted at r in $G \setminus v_d$ has been built.

Lemma 6.5.3 *After the execution of $MTCP\_MPS$ on a graph $G = (V, E)$, node r constructs a maximal tree including all the nodes reached by r in $G \setminus v_d$.*

Proof. The proof is the same as such used for the proof of the lemma about the $MTCP\_GRS$. The algorithm is based on a "depth-first search" of the on building tree. Thus, all the nodes linked to the root are explored using actions: $MTA2(2)$, $MTA2(3)$, $MTA8$ and $MTA9$. At the end all the nodes reached by the root set their variables "Stage" to $Ended$. □

Lemma 6.5.4 *Given a graph $G = (V, E)$. The $MTCP\_MPS$ uses at most $4(\#E + \#V)$ messages.*

Proof. To prove this lemma, we show an example of the run of the $MTCP\_MPS$ in the worst case.

1. The action $MTA1$ can only be applied once. So its cost is due to the application of the *marking procedure* to the nodes of the sub-tree rooted at r : At most $(2\#V_{T_r} - 2)$ messages.

2. The action $MTA2$ is applied by each node of the graph $G \setminus v_d$ such that its "Stage" variable is set to SC . The application of this action involves, in the worst case, c_1 <**mt_sc**> messages, c_2 <**mt_deleg**> messages, such that $c_1 + c_2 = \Delta - 1$, and one <**mt_ended**> message. So, in the worst case, the total cost of the action $MTA2$ is $2\#E$ messages.

3. The action $MTA3$ can be also applied only once for each node of the sub-trees rooted at the sons of v_d different from r. So its cost is due to the application of the *marking procedure* to each of these sub-trees augmented by the cost of the *root changing procedure*. The number of calls of this procedures is exactly the number of sons of v_d. The procedures cost is at most $(\sum_{i=1}^{\#Sons(v_d)}(4\#V_{T_{v_i}} - 4))$ where $v_i \in Sons(v_d)$: Adding the cost of $MTA1$, this part of the action is bounded by $4\#V - 4\#Sons(v_d)$. At each reconnection one <**mt_son**> message will be sent, otherwise one <**mt_fail**> will be sent if the reconnection failed. At most the action $MTA3$ uses $(4\#V - 4\#Sons(v_d)) + (\#Sons(v_d) - 1) + (\#E - \#V - 2 - \#Sons(v_d))$ messages.

4. The action $MTA4$ does not involve messages.

5. The action $MTA5$ is applied once by each node to update the "To_Ignore" and the "Potential" sets of its neighbors not in the tree. So, a priory, the cost of application of this action is $2\#E$, but since we use the list "To_Ignore" only $\#E$, applications are necessary. Moreover, the edges of the tree have not to be taken into account and also the sons of v_d. Finally, the cost of the use of $MTA5$ is $\#E - \#V - 2 - \#Sons(v_d)$ messages.

6. Both the actions $MTA7$, $MTA8$ do not involve messages.

7. The action $MTA9$ sends one <**mt_deleg**> message and one <**mt_ended**> message over the edges of the tree without v_d (induced by the use of the "depth-first trip"). Thus, this action involves at most $2(\#V - 2 - \#Sons(v_d))$ messages.

□

6.5.1 $2\mathcal{VCT}\_\mathcal{MPS}$ Algorithm

Now, we present an implementation of our protocol referred to in the following as $2\mathcal{VCT}\_\mathcal{MPS}$ algorithm. This algorithm uses nine actions and three procedures: The *spanning tree construction procedure* to construct the investigation tree, the *simple path propagator procedure*, the *configuration procedure* and the *maximal tree construction procedure*. Such procedures allow the implementation of phases composing the algorithm. Now, we present a short description of the two new procedures.

Simple Path Propagator Procedure ($SPPP\_MPS(T; v; Traversed, X, Y)$) The aim of this procedure is to propagate an information along a simple path linking node v to the root of a tree $T = (V, E_T)$. We will introduce the label $Traversed$ to notice the expected distinguished simple path. Initially, all the nodes have their variables "Traversed" valued X. At the end of the application of this procedure each node u in the simple path has its variable "Traversed" set to Y. This procedure is implemented using the first part of the *root changing procedure*.

Property 6.5.5 *Let $T = (V, E)$ be a tree rooted at v_0. Let v be a node in T. By executing the $SPPP\_MPS$, if v sends an information to v_0, then this information will be received by v_0 using at most $\#V$ messages.*

Configuration Procedure ($CP\_MPS(Inv\_T, T; v_0, v_d; Stage, X, Y; Required)$**)** This procedure allows us to achieve the required initializations of the *maximal tree construction procedure* and the computation of the succorer sons removing the trace of the last computations. So this procedure is executed by the root v_0 of a tree T involving all the nodes of the graph to initialize the variables used in the $MTCP\_MPS$. The initialization cycle may be implemented using an extension of the *marking procedure* without adding extra operations. Moreover, v_d prepares its disconnection, so each of its sons sets its variable "Father" to $\perp$.

Property 6.5.6 *Let v_d be a chosen node in the graph $G = (V, E)$ and let Inv_T be a spanning tree of G rooted at v_0. Starting at v_0, the CP_MPS configures T as a spanning tree of G using Inv_T and disconnects v_d. The cost of this procedure is due to the application of the marking procedure adding $\#Sons(v_d)$ <**c_tokD**> messages and $\#Sons(v_d)$ <**c_backD**> messages.*

For our case, the required initializations of the *configuration procedure* concern the following variables:

$Father(v), Sons(v), Potential(v), To\_Ignore(v), To\_Explore(v), Treated(v), Traversed(v)$.
So, we denote by $Required\_CP$ the modification to be included in the corresponding action. In other words, the use of the term $Required\_CP$ in some action is equivalent to:

1. $Father(v) := InvFather(v)$;
2. $Sons(v) := InvSons(v)$;
3. $Potential(v) := \emptyset$;
4. $To\_Ignore(v) := \emptyset$;
5. $To\_Explore(v) := Sons(v)$;
6. $Treated(v) := \emptyset$;
7. $Traversed(v) := 0$;

As mentioned above, to implement efficiently our protocol, we do the same extension of the *root changing procedure* as the one used in the local computations model. This change allows us to compute the set "Treated" during the execution of this procedure. Recall that for some disconnected node v_d, this set contains its sons included in the current maximal tree. So during the execution of RCP_MPS, at each inclusion of some node v which is son of v_d, v informs v_d sending a <**rc_included**> message in order to update its set "Treated" as follows:

$Treated(v_d) := Treated(v_d) \cup \{q\}$;

This induces that the cost of the RCP_MPS is augmented by $\#Sons(v_d)$ messages for each tested node v_d. Then the $MTCP\_MPS$ is extended, in the worst case, by $2\#E$ messages when applied to a graph $G = (V, E)$.

Let a network modeled by a graph $G = (V, E)$ such that $\forall v \in V$, $InvStage(v) = N$. The node v_0 starts the computation: It builds a spanning tree Inv_T of G rooted at v_0 ($VCTA1$).

Now, all the vertices of G are such that $InvStage = A$. Since the test scheme is based on the use of a "depth-first trip" exploration on the tree Inv_T, the root v_0 is the only one able to start the trip with the action $VCTA2$. Each node ready to be examined switches its label to D ($VCTA1(1)$, $VCTA6(1)$).

Algorithm 19 2-Vertex connectivity test algorithm ($2\mathcal{VCT}\_\mathcal{MPS}$)

- **Input:** A graph $G = (V, E)$ and a node $v_0 \in V$.
- **Variables:**
  - $Stage(v)$, $B(v)$, $Treated(v)$,
  - $InvStage(v) \in \{A, D, KO, N, OK, Over, W\}$,
  - $Traversed(v) \in \{0, 1\}$,
  - $InvFather(v)$, $InvSons(v)$.
- **Initialization:**
  - $\forall v \in V$, $InvStage(v) = N$, $Treated(v) = \emptyset$ and $Traversed(v) := 0$,
  - $\forall\, v \in V, \forall u \in B(v)\ SUC(v)[u] = false$
- **Result:** Two possible results :
  - $InvStage(v_0) = Over$ to mean that the graph G is 2-connected.
  - $InvStage(v_0) = KO$ to mean that the graph G is not 2-connected.
- **Actions:**

$\mathbf{V}CTA1$: {Node v_0 starts to build the tree $Inv\_T(InvFather, InvSons)$ }
STP_MPS($\mathbf{G}$, **Inv_T**; $\mathbf{v_0}$; **InvStage**, $\mathbf{N}$, $\mathbf{A}$; **InvFather**; **InvSons**);
1: **if** ($InvSons(v_0) \neq \emptyset$)
$InvStage(v_0) := D$;
CP_MPS(**Inv_T**, $\mathbf{T}$; $\mathbf{v_0}$, $\mathbf{v_0}$; **Stage**, $\mathbf{N}$, $\mathbf{A}$; **Required_CP**);
$Traversed(v_0) := 1$;
Let p be a first son of v_0;
send<**vct_disc**> via port p;

$\mathbf{V}CTA2$: {A message <**vct_disc**> has arrived at r from port q}
$\mathbf{Suc(r)[v_0]} := \mathbf{true}$;
MTCP_MPS($\mathbf{G}$, $\mathbf{T}$, $\mathbf{T_r}$; $\mathbf{v_0}$, $\mathbf{r}$);
send<**vct_max**> via port $InvFather$

$\mathbf{V}CTA3$: {A message <**vct_max**> has arrived at v_0 from port q}
$Traversed(v_0) := 0$;

$\mathbf{V}CTA4$: {For each node v_d such that $Stage(v_d) = D$, $Traversed(v_d) = 0$ and $Treated(v) = Sons(v_d)$ execute once:}
1: **if** ($InvSons(v_d) \neq \emptyset$)
$InvStage(v_d) := W$;
Let p be a port number in $InvSons(v_d)$;
$InvSons(v_d) := InvSons(v_d) \setminus \{p\}$;
send<**vct_tok**> via port p;
2: **else**
$InvStage(v_d) := OK$;
send<**vct_ok**> via port $InvFather(v_d)$;

$\mathbf{V}CTA5$: {For each node v_d such that $Stage(v_d) = D$, $Traversed(v_d) = 0$ and $Treated(v) \neq Sons(v_d)$ execute once:}
$InvStage(v_d) := KO$;
send<**vct_ko**> via port $InvFather(v_d)$;

$\mathbf{V}CTA6$: {A message <**vct_tok**> has arrived at v_d from port q}
1: **if** ($InvSons(v_d) \neq \emptyset$)
$InvStage(v_d) := D$;
$Traversed(v_d) := 1$
SPPP_MPS(**Inv_T**; $\mathbf{v_d}$; **Traversed**, $\mathbf{0}$, $\mathbf{1}$);
2: **else**
$InvStage(v_d) := OK$;
send<**vct_ok**> via port $InvFather(v_d)$;

$\mathbf{V}CTA7$: {For each the root v_0, such that $Traversed(v) = 1$, execute once:}
CP_MPS(**Inv_T**, $\mathbf{T}$; $\mathbf{v_0}$, $\mathbf{v_d}$; **Stage**, **Ended**, $\mathbf{A}$; **Required_CP**;
MTCP_MPS($\mathbf{G}$, $\mathbf{T}$, $\mathbf{T_{v_0}}$; $\mathbf{v_0}$);
SPPP_MPS$^{-}\mathbf{1}$(**Inv_T**; $\mathbf{v_0}$; **Traversed**, $\mathbf{1}$, $\mathbf{0}$);

$\mathbf{V}CTA8$: {A message <**vct_ok**> has arrived at v from port q}
1: **if** ($InvSons(v) \neq \emptyset$)
$InvStage(v_d) := W$;
Let p be a port number in $InvSons(v)$;
$InvSons(v) := InvSons(v) \setminus \{p\}$;
send<**vct_tok**> via port p;
else
$InvStage(v_d) := OK$;
2: **if** ($InvFather(v) \neq \perp$)
send<**vct_ok**> via port $InvFather(v)$;

$\mathbf{V}CTA9$: {A message <**vct_ko**> has arrived at v from port q}
$InvStage(v) := KO$;
1: **if** ($InvFather(v) \neq \perp$)
send<**vct_ko**> via port $InvFather(v)$;

The examination phase of the node v_d consists on the check if such a node admits a succorer son (see Theorem 6.3.3). For the particular case of the root v_0, v_0 sets its variable "InvStage" to D and applies CP_MPS to build an auxiliary tree T. Then, it chooses one of its son r in Inv_T as its succorer son and sends a <**vct_disc**> message to such a son($VCTA1$). When a node r receives a <**vct_disc**> message from its father v_0, it sets its corresponding variable "SUC" to $true$ and starts to build a maximal tree of $G \setminus v_0$ applying the $MTCP\_MPS$.

For the other cases, each node proceeds as follows (action $VCTA6(1)$): The node v_d labeled D informs the root of the investigation tree about its attention applying the $SPPP\_MPS$. When the root is informed, it applies action $VCTA7$: It configures an auxiliary tree T which is the same as the investigation tree with the disconnection of v_d. Then, it builds a maximal tree of $G \setminus v_d$ applying the $MTCP\_MPS$. When the root v_0 built its maximal tree, it responds to v_d applying the $SPPP^{-1}\_MPS$. For the particular case of the root, the chosen son r informs v_0 that it has extended its tree as much as possible sending a <**vct_max**> message.

When v_d receives such a response, it starts to compute its succorer sons: It is based on the set "Treated" computed during the execution of the RCP_MPS induced by the execution of the $MTCP\_MPS$. There are two possibilities.

First, node v_d finds one of its sons not included in the constructed maximal tree(action $VCTA5$). So, v_d fails its test and sets its variable "InvStage" to KO sending a <**vct_ko**> message to its father in Inv_T. At the reception of such a message, node v propagates this message to the root applying the action $VCTA9(1)$. When v_0 receives such a message, it states that G is not 2-connected.

Otherwise, node v_d discovers that all its sons are included in the maximal tree of the root (or of the chosen son of the particular case of the root). Then, it continues the trip, if possible, sending a <**vct_tok**> message to one of its not yet explored son in the investigation tree ($VCTA4(1)$). Otherwise, it detects that all the nodes in its sub-tree are visited. In this case, it sets its variable "InvStage" to OK and informs its father sending a <**vct_ok**> message to continue the exploration ($VCTA4(2)$).

A node which receives a <**vct_tok**> message becomes the node to be examined($VCTA6$). A node which receives a <**vct_ok**> message looks in its sons if there is a not yet explored son applying the action $VCTA8$. Furthermore, only the root v_0 detects the end of the "trip" when it receives a <**vct_ok**> message from its last son: In this case, it affirms that the graph G is 2-connected.

6.5.2 Overall Proofs and Complexities Analysis of the $2\mathcal{VCT}\_\mathcal{MPS}$

Now, we show the correctness of the $2\mathcal{VCT}\_\mathcal{MPS}$ algorithm using a scheme based on the procedures properties as explained in the section about the system model. The analysis is closed with time complexity measures.

Theorem 6.5.7 *The $2\mathcal{VCT}\_\mathcal{MPS}$ algorithm presented above implements a distributed test of the 2-vertex connectivity of a graph.*

Proof. The proof follows the run presented above and use the same scheme as the one used to prove the $2\mathcal{VCT}\_\mathcal{GRSS}$ algorithm. □

Lemma 6.5.8 *Given a graph $G = (V, E)$. The $2\mathcal{VCT}\_\mathcal{MPS}$ algorithm tests the 2-vertex connectivity of G using at most $O(deg(g) \times \#V^2)$ messages.*

Proof. The worst case of the test algorithm corresponds to the case when the *maximal tree construction procedure* is applied for all the nodes. So we will detail the run of the $2\mathcal{VCT}\_\mathcal{MPS}$ algorithm for this case.

1. The action $VCTA1$ is applied once. Its cost is due to the application of the *spanning tree procedure* to the graph G to build the investigation tree Inv_T, followed by the *configuration procedure* to build an auxiliary tree T to test v_0 and the sending of a message to its succorer son.

2. The action $VCTA2$ is applied once by the succorer son to build its maximal tree in spite of the disconnection of the root. Its cost is due to the application of the $MTCP\_MPS$ plus one message.

3. The action $VCTA3$ does not involve messages.

4. The action $VCTA7$ is applied by the root one time for each tested node. It involves the execution of the CP_MPS, $MTCP\_MPS$ and $SPPP^{-1}\_MPS$.

5. The actions $VCTA4$, $VCTA6$, $VCTA8$ are used to implement the "depth-first trip" on the tree $Inv\_T = (V, E_T)$. So, the cost of such an exploration is $2\#V - 2$ messages. In addition, the action $VCTA6$ involves the application of the $SPPP\_MPS$.

Without loss of generality we denote by V the set of the nodes and by E the set of edges of all the graphs structures used in all the procedures. From the previous, we can claim the following:

- $COST(STP\_MPS(G, T; v_0; Father; Sons; Stage, X, Y), messages) = 2\#E + \#V - 1$ (see Property 6.2.7).
- $COST(MTCP\_MPS(T; r), messages) = 6\#E + 4\#V$ (see Lemma 6.5.4).
- $COST(CP\_MPS(G, T), messages) = 2\#V - 2$ (see Property 6.5.6).
- $COST(SPPP\_MPS(T; v; Traversed, X, Y), messages) = \#V - 1$ (see Property 6.5.5).
- $COST(SPPP^{-1}\_MPS(T; v; Traversed, X, Y), messages) = \#V - 1$ (see Property 6.5.5).

Therefore, the total number of messages exchanged during the execution of the $2\mathcal{VCT}\_\mathcal{MPS}$ algorithm is at most $2\#E(3\#V + 1) + 7\#V^2 + V - 2$ messages. Then, the message requirement of the $2\mathcal{VCT}\_\mathcal{MPS}$ algorithm is in $O(deg(G) \times \#V^2)$. □

For the space complexity, we used the same data as those used to encode the $2\mathcal{VCT}\_\mathcal{GRS}$, then we claim the following:

Lemma 6.5.9 *The space requirement of $2\mathcal{VCT}\_\mathcal{MPS}$ algorithm is in $O(deg(G) \times \log deg(G))$ bits per node when the algorithm is applied to a graph $G = (V, E)$.*

6.5.3 Time Complexity

To analyze the time complexity of algorithms based on graph exploration on asynchronous networks, we use a level number and a diameter defined in Section 2.1 to help us in the proofs of the worst case bound. We associate with leaves of a tree T level $l_0 = 0$ and the level of each node is the maximum level number of its sons plus 1. This number will be computed during the construction of the investigation tree. So the message <**st_back**> is replaced by the level number. We introduce the function $To\_number(msg)$ (resp. $To\_message(level)$) to convert the <**st_back**> message (resp. the level number) to a level number (resp. to a <**st_back**> message). Thus, actions $STA2$ and $STA4$ of the *spanning tree procedure* are modified such as :

- The level number $level(v)$ is initialized to 0 in $STA2(1)$. Then, the message $To\_message(level(v))$ is used as a substitute for <**st_back**> message.

- **S**$STA4$: {A message <**st_back**> has arrived at v from port q}
  $Terminated(v) := Terminated(v) \cup \{q\}$;
  $level(v) := max(level(v), To\_number($**st_back**$)$;
  1: **if** $(Included(v) = B(v)$ and $Terminated(v) = Sons(v))$
  $Stage(v) := Y$
  $level(v) := level(v) + 1$;
  2: **if** $(Father(v) \neq \perp)$
  send<**To_message(level(v))**> via port $Father(v)$;

The two following lemmas follow the definition and the computation of the level:

Lemma 6.5.10 *Given a graph G with diameter $diam(G)$, the maximum node level number assigned during the execution of the STP_MPS is bounded by $diam(G)$.*

Proof. Initially, let T_0 be the tree containing the leaves of the tree to be build. Every leaf node v_l has level $level(v_l) = 0$. At this step, the leaves are discovered (action $STA2$). The tree T_l is obtained after the local termination of the nodes of level $l - 1$. This step involves the send of their level numbers (action $STA4(2)$). So, we can do this at most $diam(G)$ times before we get a spanning tree of G. Thus, the maximum level number obtained is $diam(G)$. □

Lemma 6.5.11 *During the execution of the STP_MPS on the graph G, the node at level l will have sent a* **st_back** *message within $diam(G) + l$ time.*

Proof. Let t_0 be the time at which all the nodes of G are included in the on-building tree T (labeled WA). Now, by induction we show that for each node v at level l in the graph, v will sent a <**st_back**> message within $t_0 + l$ time.

For the case of $l = 0$, we observe that the nodes (leaves) at level 0 have sent their <**st_back**> message either before or at time t_0. Since at time t_0 all the nodes are already included in the on-building tree T, the lemma is true. Let u be a leaf of the tree T. If u is included before t_0, it has already sent its <**st_back**> message. If u is included at time t_0 (the worst case), then the <**st_back**> message is sent immediately after its inclusion (action $STA4(2)$). Therefore, it is true that a node at level 0 will have sent its <**st_back**> message within time $t_0 + l$, where l is the level number of the node.

Now, we suppose that the lemma is true for level number $l - 1$. It means that all the nodes at level $l - 1$ have sent their <**st_back**> messages either before or at time $t_0 + l - 1$. Now, we show that all the nodes at level l will have sent their <**st_back**> messages within time $t_0 + l$.

During the construction of the spanning tree, each included node has only one defined father. A node at level l must have at least one neighbor (son) at level $l-1$. Let u be a node at level l. From the induction hypothesis, all the neighbors (sons) of u at level $l-1$ or less had sent their <**st_back**> messages within time t_0+l-1. According to the assumption about the time complexity, which stipulates that the message delay is at most one time unit, the <**st_back**> messages from the sons of u at levels $\leq l-1$ will arrive at u within time $(t_0+l-1)+1$. Since u has at most one neighbor at level $\geq l$ (father), now u is ready to send its <**st_back**> message when it receives such a message from all its sons at level $\leq l-1$. Therefore, node u at level l will have sent its <**st_back**> message within time t_0+l. □

From the two previous lemmas, we deduce:

Lemma 6.5.12 *The execution of the STP_MPS on the graph G uses at most $2diam(G)$ time.*

Using the same proofs techniques, we claim the following:

Lemma 6.5.13 *Given a tree $T=(V,E)$ with diameter $diam(T)$, the MP_MPS on T uses at most $2diam(T)$ time.*

Lemma 6.5.14 *The execution of the RCP_MPS on a tree T with diameter $diam(T)$, requires at most $2diam(T)$ time.*

Now, we study the time complexity of the main procedure: $MTCP\_MPS$. We use the same reasoning as such used for the message complexity.

Lemma 6.5.15 *Given a spanning tree $T=(V,E_T)$ of a graph $G=(V,E)$ rooted at v_0. Let v_d be a node in G, then the maximal tree construction procedure uses at most $6\#V-2\#V_{T_{v_0}}$ time to build, if possible, a spanning tree of $G\setminus v_d$, where T_{v_0} is the sub-tree rooted at v_0, at the beginning of the $MTCP\_MPS$.*

Proof. First, v_0 applies the MP_MPS to construct the base of the "on-building" tree. Second, we know that each sub-tree resulting after the disconnection of v_d is traversed sequentially because of the use of the "depth first" exploration technique. So without loss of generality, each sub-tree applies one time the RCP_MPS followed by the MP_MPS. Let u be a son of v_d. The number of traversed edges is bounded by $2(\#V-2)$ for the reconnection phase. Thus, the total time used by the $MTCP\_MPS$ is a priory bounded by $4(\sum_{i=2}^{\#Sons(v_d)} diam(T_{v_i}))+2diam(T_r)+2(\#V-2)$, such that $v_i \in Sons(v_d)$. The sum of the diameters of all the sub-trees is bounded by $\#V$, then, the *maximal tree construction procedure* requires time bounded by $6\#V-2\#V_{T_r}-4$. □

Lemma 6.5.16 *Given a tree $Inv\_T=(V,E_{Inv\_T})$, the "depth first trip" on Inv_T uses at most $2(\#V-1)$ time.*

Proof. Since each edge of the tree is traversed two times and $\#E_{Inv\_T}=\#V-1$, the total time required to traverse a tree T using "depth first trip" technique is $2(\#V-1)$. □

The following result summarizes the complexity analysis.

Theorem 6.5.17 *Given a graph $G = (V, E)$. The distributed algorithm presented above computes the test of the 2-vertex connectivity of G in $O(\#V^2)$ time using $O(deg(G) \times \#V^2)$ messages and $O(deg(G))$ bits per node.*

6.6 Status and Future Works

This work deals with the test of the 2-vertex connectivity of graphs in a distributed setting using local knowledge. We present a new formalization of such a problem using the notion of succorer sons. This notion is defined under an arbitrary pre-constructed spanning tree, of the graph to be tested, called the investigation tree. So using a constructive approach we compute the set of succorer sons of all the nodes. Therefore, to check whether a given graph is 2-connected it suffices that the only node which admits a succorer son is the root of the investigation tree.

The protocol is encoded in the local computations model and implemented in the message passing model using a set of procedures. Given a graph $G = (V, E)$ with degree Δ and N nodes: The first algorithm requires $O(\Delta \times N^2)$ steps and $O(deg(G) \times \log deg(G))$ bits per node. For the second one, tacking into account the extra actions used to call the procedures, to collect the result of the computation and without combining some phases to reach better some constants, the distributed algorithm presented above achieves correctly the test of the 2-vertex connectivity of G in $O(N^2)$ time using $O(\Delta \times N^2)$ messages and $O(deg(G) \times \log deg(G))$ bits per node.

Furthermore, our work has an extra benefit: Algorithms encoded in the used version of local computations model may be implemented in the asynchronous message passing model with a transformation including simple changes. Furthermore, the transformation guarantees the following: (1) the proofs used in the first one allows to deduce the proofs for the second one, (2) the complexity measures may be also deduced.

Now, we propose some further research to extend our works. For the use of the procedures to encode distributed algorithms: They simplify their design and their study. But the assumptions about their atomic applications may violate the performances of the designed applications. So it is useful to propose a general methodology for proving their "correctness calls" without such assumptions. This is itself a research topic and the present work is a motivating example.

For the k-vertex connectivity test, to our knowledge there is no distributed algorithm based on local knowledge to deal with this problem. So, we propose two possible implementations: First, we can imagine the definition of a total ordering on vertices, based on a spanning tree with an ordering on outgoing edges; and then a systematic enumeration of k-tuples of vertices, to check whether they separate. This would not be efficient, but would indicate a theoretical possibility. Second, we conjecture that we can use the previous procedures to generalize our algorithm to test the k-vertex connectivity of graphs in polynomial time. Intuitively, we can reduce this problem to the case of $k = 2$ followed by an incremental test procedure.

One interesting application of the distributed k-vertex connectivity test is the measure of a network fault-tolerance in a decentralized environment (for example an ad hoc network without a GPS satellite). In fact, the quality of the reachability of any couple of nodes in an unreliable network, and hence their communication, will depend on the number of paths

between these nodes. This number of paths is the connectivity. If such paths are already computed, it is essential to maintain the reachability between nodes while crash or disconnection of certain nodes occurs.

We can also determine by an appropriate labeling the 2-connected components: For each 2-connected component, choose a "leader edge", and encode procedures that for any edge will say who is the leader of its 2-connected component. Then, for 2 edges determine whether they belong to the same component. If they are not, another procedure should determine a path from one to the other. In this way, we can construct the tree of 2-connected components. Note that the problem of finding k connected sub-graph with minimum cost is known to be $NP-$ hard. For more examples the reader is referred to [BHM02, CW04, Wes01].

Chapter 7

Distributed Maintenance of Spanning Trees

MAintainability refers to how easy a failed system can be repaired. Many applications in distributed environments are based on the network topology or structure knowledge. The designed protocols often used this knowledge as input information. In our case, the knowledge is the vertex connectivity applied to design protocols on unreliable networks modeled by graphs. For the routing applications [CW04], each node uses information about its neighborhood to update the routing tables when the topology changes. So, it is suitable to maintain dedicated structures to guarantee the perennity of the routing. An important measure in this routing schemes is the number of nodes (or processors) to be updated upon a topology change. In our context, we use a spanning tree of the network described as a graph: A distributed system is represented as a connected, undirected graph denoted by $G = (V, E)$ where a node in V represents a process and an edge in E represents bidirectional communication link. Let M denotes the number of its edges, N denotes the number of its nodes and Δ denotes its degree.

The studied spanning tree maintenance problem is stated as follows. After the crash of a node v, we have to rebuild another spanning tree of G deprived of v with a minimum of changes. Obviously, if G becomes disconnected this fails. In this case, the graph modeling the network is splited into a multiple connected components. Therefore, we consider a maintenance of a forest of a spanning trees. That is, how to associate a spanning tree with each of these components. It is assumed that a crash detection service is implemented using unreliable failure detectors [CT96, HM05b], satisfying: After the crash of some node, the crash detection service reaches a stable state and then all the neighbors are informed about the crash. The stability of the crash detection service is assumed to be reached in a finite time after the stability of its failure detectors. We consider a decentralized setting where nodes may crash by permanently halting and can perform only computations based on local knowledge.

In this chapter we propose a new formalization of the problem of the maintenance of a forest of spanning trees, as a general occurrence, and the spanning tree maintenance for k-connected graphs as a specific occurrence. At each step of maintenance executed by our protocol after each failure occurrence, we would like to preserve the maximal possible structure of the existing spanning trees. That is, in order to minimize the number of nodes to be changed. We extend the notion of *succorer sons* introduced in Chapter 6. Here, such information will be

used as a basic local knowledge used by the processes to take local decisions in order to maintain a desired structure of a graph in the presence of crash failures. Recall that a succorer son of a node v in T is a son which is able to build a spanning tree of some connected component after the deletion of v.

After the crash of some node, if the graph remains connected then it suffices to activate its succorer son to rebuild a spanning tree. In the opposite case, the graph is decomposed in multiple components. Thus, our protocol uses succorer sons information associated with each node achieved using a pre-processing task to identify the set of sons to be activated. The protocol is divided into three distinct parts: (1) the detection and the propagation of the failure information, (2) maintenance of a spanning tree of each resulted connected components and (3) update of the succorer sons information. For a given graph G, using the local computations model: To deal with each failure occurrence, without taking into account the delay caused by the used failure detection service, the first part uses $O(N)$ time. The second part uses $O(\Delta \times N)$ steps and the last part takes $O(\Delta \times N^2)$ steps. The space complexity is in $O(\Delta \times \log \Delta)$ bits per node. The protocol is applied concurrently to each of the resulted disconnected component. As we will see, we proposed also an optimal version of this protocol for k-connected graph in the presence of at most $k-1$ failure occurrences.

The outline of the chapter is as follows. We start with a few survey of existing solutions in Section 7.1. In Section 7.2, we describe our system model including the set of procedures that we will use in our formalization. Section 7.3 presents our approach to deal with the problem of maintaining of a forest of spanning trees using the notion of succorer sons. Then, Section 7.4 shows a method to compute the succorer sons labeling of a graph. Section 7.5 discusses the update of such a labeling after crash failure occurrence. In Section 7.6 we present an encoding of the proposed protocol in the local computations model. An overall proofs of its correctness and its analysis is given in Section 7.7. Then, in Section 7.8 we focus on efficiency: We show how it is possible to profit from the network topology. Section 7.9 concludes the chapter with the presentation of our findings and a short discussion about extended works.

7.1 Related Works

The specific protocols devoted to maintain a spanning tree are proposed in [ACK90, AS97, Por99, RSW05, GA05]. In [ACK90], the authors studied the tree maintenance problem as the graph's topological changes after edge failures. The maintained spanning tree is obtained using incremental update procedures, the time complexity of their solution is $O(\Delta \log M + N)$.

In [AS97], an an algorithm to maintain a common data in a communication network is presented. It operates in a communication network where nodes are arranged in a chain. The solution is an improvement of some previous solutions including those based on *Full Broadcast* and on *Broadcast with Partial Knowledge* [ACK+91]. The time complexity in both algorithms is $O(N + M)$.

Therefore, the solution presented in [AS97] solved the *Broadcast with Partial Knowledge* problem in $O((M+N)\log^3 M)$. The protocol presented in [Por99] is composed of iterations to rebuild a spanning tree of a maximal connected component when an edge fails. The time complexity of such a protocol is in $O(\text{“}actual\ size\text{”})$ of the biggest connected component adding an extra polylogarithmic time.

Nevertheless these algorithms, either use complete networks, or use identities for nodes

and deal with edge failures. Moreover, the notification service of failures are assumed to be reliable. In [RSW05], the last assumption is not taken into account and node failures are considered. They, as described previously, use stable failure detectors. In order to encode such an algorithm, a structure called "expander graph" is implemented. This structure is more expensive than a simple spanning tree construction.

[GA05] presented a protocol based on the use of Nevilles's code to maintain a spanning tree of a complete graph. In this work, the maintenance task is formalized as a set of constraints to be checked. For the complexity analysis, they assume that a node can perform operations on $O(N)$ variables in $O(1)$ time. So the algorithm is in $O(N)$. Such assumption steels very far from the real behaviors in distributed environments. Most of these works are presented in a centralized setting or assuming global knowledge about the graph to be examined.

7.2 The System Model

In this chapter, we consider node failures in arbitrary network modeled by a graph. When starting our protocol the graph is assumed to be connected. Our protocol uses only local knowledge. That is, to perform a computing step, only information related to the states of the neighbors is required. Especially, each node knows its neighbors in G and in the current spanning tree T the "position" of a node is done by its $father$ except for the root, and a set of ordered $sons$. Furthermore, the studied network is *semi-anonymous*: Only the root needs to be identified. Moreover, we investigate some applications that can use our protocol.

Here, processes communicate and synchronize by sending and receiving messages through the links. There is no assumption about the relative speed of processes or message transfer delay, the network is *asynchronous*. Each node communicates only with its neighbors. The links are reliable and the processes may fail by crashing. To encode our algorithms, we will use mainly graph relabeling systems model. The message passing model will be used to demonstrate that the designed protocol is general. For the main model, we consider only relabeling between two neighbors. That is, each one of them may change its label according to rules depending only on its own label and the label of its neighbor.

Now we give the list of formal procedures that we will use to build our protocol. Except the first one which will be described below, the others are already detailed in Chapter 6.

1. *crash detection procedure* ($CDM\_GRS(G; Crashed)$)
2. *spanning tree procedure* ($STP\_GRS(G, T; v_0; Stage, X, Y; Father; Sons)$)
3. *marking procedure* ($MM\_GRS(T; v_0; Stage, X, Y)$)
4. *root changing procedure* ($RCM\_GRS(T; r)$)
5. *simple path propagator procedure* ($SPPM\_GRS(T; v; Traversed, X, Y)$)
6. *maximal tree construction procedure* ($MTCM\_GRS(G, T, T'; v_d, r)$)
7. *configuration procedure* ($CM\_GRS(Inv\_T, T; v_0, v_d; Stage, X, Y; Required)$)

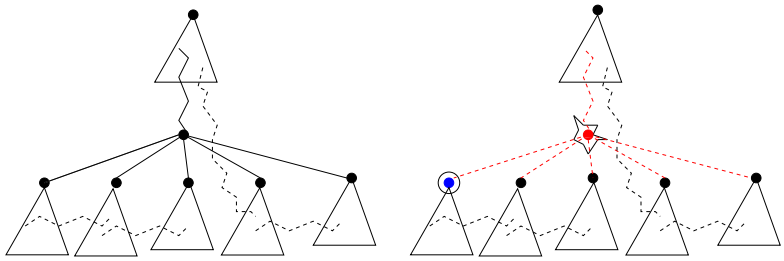


Figure 32: Succorer sons(a,b)

7.2.1 Crash Detection Procedure ($CDP(G; Crashed)$)

The *crash detection procedure* is based on an unreliable failure detection service. Such a service may be implemented in asynchronous distributed systems using timeout and heartbeat strategy for example (see Section 3.4 or for more details [CT96, HM05b]). The CDP informs, only after the stability of its corresponding failure detector, each node about the possible crash of one of its neighbors: The failure detector of any non crashed node v has detected all the crashed neighbors which are now stored in a list of crashed neighbors $Crashed(v)$. Moreover, all the sets used in the labels will be devoted from crashed processes. For example, if the father of a node v is in $Crashed(v)$, then it is set to $\perp$. Thus, we denote by GST [1] the time after which the failure detectors of all nodes stabilize. We assume that the execution of one rule requires one time unit as the time unit to measure the failure detector stability. Therefore, the cost of the CDM is in $O(GST + deg(G))$ for each failure occurrence.

7.3 Maintenance of a Forest of Trees

As the connectivity of a network might change over time, the graph modeling it may be disconnected. In fact, after the deletion of some nodes, results of failures, the network may become splited and partitioned into many components (see Figure 32.b). It would be desirable to maintain a dynamic structure of trees in any graphs. In this section, we deal with the maintenance of a forest of trees of a graph which is initially k-connected where $k \geq 1$ and any number of failures. That is, each component computes its spanning tree. Therefore, we need to give to each node the label SUC to encode its ability to construct a spanning tree of its component after the failure of its father. Thus, such a labeling increases the degree of the parallelism during the computation. Indeed, succorer sons update the spanning tree of their components concurrently. As shown in Figure 32.b, the crashed node is the one in the star and its succorer son is the one in the circle.

7.3.1 The Labeling

To describe the label SUC we use the following notations: If u has degree d, we denote by $v_1, v_2, \cdots v_d$, its neighbors such that $v_1 < v_2 < \cdots < v_d$, then SUC is a vector of Boolean and to encode the information that u has about v_i we use $SUC(u)[v_i]$. Now we present a definition that we will use to specify the properties that the labeling SUC must satisfy.

[1] GST for Global Stabilization Time.

Definitions 7.3.1 *Let $G = (V, E)$ be a connected graph and T be a spanning tree of G rooted at v_0. For each pair of nodes u, v in G, we say that u is reached by v in G iff there is a path in G linking v to u. After the deletion of a node v_f in V, G is splited into DC disconnected components. Recall that in T, the set $Sons(v_f)$ denotes the set of ordered sons of v_f. Then,*

1. *We say that v is a succorer son of v_f iff the following holds:*

   (a) $v \in Sons(v_f)$,

   (b) if $v_f \neq v_0$ *then* v *is not reached by* v_0 *in* $G \setminus v_f$,

   (c) $\neg\exists\, u \in Sons(v_f)$ *such that* $u < v$ *and* v *is reached by* u *in* $G \setminus v_f$.

   Then, a label $SUC(v)[v_f]$ is set to $true$. Otherwise it is set to $false$.

2. *Each of the succorer sons of a node v_f, and the root of the current spanning tree of G in the case of $v_f \neq v_0$, is called a maintainer node.*

3. *We say that a protocol $\mathcal{P}$ succors G using a labeling SUC iff $\mathcal{P}$ rebuilds a forest of trees of $G \setminus v_f$. That is, $\mathcal{P}$ constructs a spanning tree of each one of the DC component. Then, SUC is called a succorer labeling of G.*

In the following, each of nodes in charge to update the forest of trees after the deletion of any node is called maintainer node. Thus, each of the succorer sons of a node v becomes a maintainer node after the deletion of v. Now, we present our distributed algorithm to deal with the maintenance of a forest of trees.

7.3.2 Maintenance of a Forest of Trees Algorithm ($\mathcal{MOFST}$)

We present an informal description of our distributed protocol dedicated to maintain a forest of spanning trees for any connected graphs. Then we give an implementation of the algorithm using means of local computations, the proof of its correctness and some complexity measurements. During the design of our algorithm we added the constraint to preserve as much as possible the existing spanning trees. For a sake of clarity, our protocol is divided into two parts: (a) the pre-processing task and (b) the maintenance invoked at each failure occurrence. To carry out the first part (a) we use the following phases:

1. *succorer sons computation.* Nodes of G compute their succorer sons. This is based on a simulation of failure.

2. *cleaning.* To deal with the failure occurrence using the same procedures and labels as those used during the simulation. For example, all the nodes will set their labels $Stage$ to A and so on.

In phase one, we use a pre-processing task to compute a labeling that will be used by the protocol to compute a spanning tree of each component. Having such a labeling, the protocol is ready to work correctly for one failure occurrence. To use the same task as used during the previous phase, phase two allows us to do required initializations of the used labels in each phase as that used in phase one.

The second part (b) is more delicate, it is composed of the following:

1. *detection of a crash.* Neighbors of the crashed node are informed that the node v_f is crashed.

2. *propagation.* Father of the crashed node, if it exists, informs the root of the tree that some node in its descendant is crashed.

3. *spanning tree maintenance.* Node u, a spanning tree maintainer, starts the updating of the spanning tree and the succorer labels of its component.

4. *cleaning.* To deal with the next failure occur using the previous five phases, the used labels require some initialization in each connected component. This is the same phase as phase 2.

5. *succorer sons computations.*

6. *cleaning.*

For part (b), phase one is done using a *crash detection procedure* as described previously. After each failure occurrence, phase two is invoked to activate the root of the tree, if its exists. In this case, it is considered as a spanning tree maintainer. Now, all the maintainer nodes may work concurrently to update (or to maintain) the spanning trees of their components in phase three. In order to compute the new succorer sons to treat the next failure occurrence, phase four and phase five are applied to each of the resulted components. Then, phase six is used to clean the used data to treat the current failure occurrence. Note that part (b) is performed concurrently in each of the resulted connected components.

7.4 Succorer Sons Computation

We propose an algorithm to compute the vector SUC for each node of a graph G. In the sequel this task is realized using an algorithm referred to as a $\mathcal{SUC}$ algorithm. As we shall see, in spite of its cost, such a task does not violate the performance of our algorithm since it is considered as a pre-processing task. That is, it allows to compute vital information pertaining to the update of the spanning tree after the network changes. Our distributed computation of succorer vectors algorithm consists of the following phases: (1) the computation of the spanning tree called *Investigation tree*, denoted by Inv_T, of G with root v_0, (2) exploration of Inv_T to compute succorer sons of all the nodes of G.

In phase one, we use an algorithm as described in the previous section. This procedure constructs a spanning tree Inv_T of a given graph G rooted at v_0 with local detection of the global termination. It means that v_0 detects the end of the spanning tree computation of G. In phase two, we explore the tree Inv_T using "depth-first trip" [Tel00] . When the trip reaches a node v_d, v_d does the following:

1. *configures.* The aim of this phase is to prepare the application of the succorer sons computation of some node v_d. So, this phase initializes the set of labels erasing the trace of the last succorer computation. This phase is done using the *configuration procedure* (see bellow).

2. *disconnects itself.* Node v_d disconnects itself, it will be labeled $Stage(v_d) = D$.

3. *computes the set of its succorer sons.* After the execution of the two previous phases, node v_d is ready to compute its succorer sons. So, v_d applies the *succorer node procedure* that we will detail in the following.

Succorer Node Procedure ($SNM(G, T; v_d)$). This procedure is applied by each observed node v_d in order to compute its succorer sons. Since we will use such information to update the spanning trees after some failures occur and we would like to preserve the maximal possible structure of the existing spanning trees, the trees rooted at the "old" roots are preserved. So, each of the updating spanning trees will use such a tree as a base for its possible extension. The computation of the succorer sons is composed of the following phases:

1. *disconnected node v_d is not the root*
   (a) *propagates.* Node v_d informs the root of the actual tree T about its disconnection.
   (b) *maximal tree construction.* When the root v_0 is informed, it starts the computation of its maximal tree.
   (c) *responds.* Eventually, v_0 constructs its maximal tree, so it responds to v_d.
2. *succorer sons computation.* Node v_d starts the computation of its succorer sons: It chooses one of its son r not included in the maximal tree constructed by the root to become its succorer son. Then, each chosen node does the following:
   (a) *new succorer son.* Node r is a succorer son of v_d. So it updates its labeling: $SUC(r)[v_d]$ is set to $true$.
   (b) *maximal tree construction.* Chosen node r is in charge to construct and to mark its maximal tree including all the nodes not reached by the above marking using the same computations as the root of T.
   (c) *all the sons of v_d are marked.* Node v_d finds that all its sons are included in some constructed trees. Therefore, the computation of its succorer sons is terminated.
   (d) *some son of v_d is not marked.* Node v_d detects the end of the current *maximal tree construction* such that there is some son of v_d not yet included. It chooses such a son as its newly chosen son to become its next succorer son and so on the computation follows.

7.5 Updating After Some Failure

In this section, we discuss the incremental maintenance of a forest of spanning trees and the update of the succorer sons information in dynamic networks. We consider the changes that are the results of the deletion of some node which may cause G to become disconnected. As mentioned above, when the $\mathcal{MOFST}$ is introduced, each of the maintainer nodes is able to rebuild a spanning tree of its component. Recall that, after the crash of a node v_f, the set of maintainer nodes is composed of the set of the succorer sons of v_f augmented with the root, if it's not v_f. We denote with *maintenance procedure* the algorithm based on *maximal tree construction procedure*. Therefore, for a graph G, after the detection of a crash of v_f it suffices that each corresponding maintainer node applies the *maintenance procedure* to maintain the forest of trees of $G \setminus v_f$. One way to update the set of succorer sons is the re-application of the $\mathcal{SUC}$ algorithm on each component.

7.6 Encoding of the $\mathcal{MOFST}$ Algorithm

In this section we propose an encoding of the $\mathcal{MOFST}$ using graph relabeling systems described in Section 1.2. So, the needed procedures are encoded in the same model and the analysis is done according to the given assumptions of such a model. We start by describing its main phase: The *succorer sons computations*.

7.6.1 Succorer Sons Computation

Now, we present an implementation of the $\mathcal{SUC}$ algorithm using six relabeling rules and invoking the two procedures: *configuration procedure* described in the previous Chapter, and *succorer node procedure* which we will describe below. Since this algorithm uses some procedures, the proof of its correctness and its complexity measures are differed after the study of the used procedures.

Succorer Node Procedure ($SNP\_GRS(G, T; v_d)$) Now, we present an implementation of the *succorer node procedure*. As we shall see, this procedure use the *simple path propagator procedure* to implement the phase $1.(a)$ and its reversible version to implement $1.(c)$. The phase $1.(b)$ and $2.(b)$ are achieved using the *maximal tree construction procedure* presented above.

Let $G = (V, E)$ be a graph and let $T = (V, E_T)$ be a spanning tree of G. We consider a node v_d as the observed node. Initially, $\forall v \in V(G)$, $Stage(v) = A$. The node v_d initiates the computation : since in G it is the only one that executes the rule $SNR1$. It informs the root v_0 about its attention to compute its succorer sons applying the $SPPP\_GRS$ ($SNR2$.) Then, node v_d waits the response from v_0. For the particular case of the root, the information phase becomes short ($SNR2$). So, in all cases node v_d will relabel itself $Stage(v_d) = Again$ ($SNR1$ or $SNR3$). Then v_d looks in its sons if there is at least one node r labeled $Stage(r) = A$. While there is such a node, v_d orders to r to construct its maximal tree, rooted at itself, of the graph $G \setminus v_d$ ($SNR4$). We denote this tree T_r. This work will be achieved using the $MTCP\_GRS$. The first succorer son r of v_d starts the execution of the $MTCP\_GRS$ and v_d waits the result of such a procedure. Eventually, r will build its maximal tree T_r, then r updates with rule $SNR5$ the set of succorer sons of v_d. So, r will be considered as a succorer son of v_d. Moreover, r informs v_d which then can proceed to order another son not included in T_r. Furthermore, v_d collects the states of its sons to know each of them are already included in some maximal trees. Recall that after the execution of the $MTCP\_GRS$, all the nodes included in T_r are labeled $Ended$. So, the observed node v_d updates its set $Treated$ to include such nodes (Rule $SNR6$). Finally, v_d detects the end of the succorer labeling computation when all its sons are included in some trees (labeled $Over$). When $Stage(v_d) = Over$ maximal trees of $G \setminus v_d$ rooted at succorer sons of v_d have been computed ($SNR7$).

Lemma 7.6.1 *Let $G = (V, E)$ be a connected graph and let T be a spanning tree of G rooted at v_0. The succorer node procedure (SNP_GRS) computes correctly the succorer sons of any node of G.*

Proof. We must prove the following property. For any node v, all its sons u_i such that $SUC(u_i)[v] = true$ constructs a maximal tree and the union of such trees and the tree of the root, if $v \neq v_0$, includes all the nodes of $G \setminus v$. We prove this property by induction on the number of succorer sons of v. We denote by $SUCS(v)$ the ordered set of sons u such that $SUC(u)[v] = true$. By induction on the size of the set $SUCS(v)$. For the case of $\#SUCS(v) =$

Algorithm 20 Succorer node procedure ($SNP\_GRS(G, T; v_d)$)

- **Input:** A graph $G = (V, E)$ with a spanning tree $T = (V, E_T)$, a node v_d to be disconnected and observed.
  - **Labels:**
    - $Stage(v) \in \{A,\ D,\ Ended,\ F,\ KO,\ OK,\ Over,\ SC,\ W\}$, $Father(v)$, $Sons(v)$, $B(v)$
    - $To\_Explore(v)$, $Potential(v)$, $Treated(v)$
    - $SUC(v)$
  - **Initialization:**
    - $\forall\, v \in V \setminus \{v_d\},\ Stage(v) = A.$
    - $Stage(v_d) = D.$
    - $Sons(Father(v_d)) = Sons(Father(v_d)) \setminus \{v_d\}.$
    - $\forall\, v \in Sons(v_d),\ Father(v) = \perp\ .$
    - $\forall\, v \in V,\ To\_Explore(v) = Sons(v).$
    - $\forall\, v \in V,\ Potential(v) = \emptyset.$
    - $\forall\, v \in V,\ To\_Ignore(v) = \emptyset.$
    - $\forall\, v \in V,\ Treated(v) = \emptyset.$
    - $\forall\, v \in V, \forall u \in B(v),\ SUC(v)[u] = false.$
- **Results:** The set of succorer sons of v_d is computed. It is composed of the sons u of v_d such that $SUC(u)[v_d] = true$.
- **Rules:**

$SNR1$: **Node v_d starts the computation** [2]

Precondition :
- $Stage(v_d) = A$

Relabeling :
- **if** ($Father(v_d) \neq \perp$) $\mathbf{SPPP\_GRS(T; v_d; Traversed, 0, 1)}$ $Stage(v_0) := W$
- **else** $Stage(v_d) = Again$

$SNR2$: **Node v_0, the root, starts its computation and marking of its maximal tree, and then it informs v_d**

Precondition :
- $Stage(v_0) = W$
- $Father(v_0) = \perp$

Relabeling :
- $\mathbf{MTCP\_GRS(G, T; v_d, v_0)}$
- $\mathbf{SPPP\_GRS^{-1}(T; v_0; Traversed, 1, 0)}$

$SNR3$: **Node v_d, is informed that v_0 has built and marked its maximal tree**

Precondition :
- $Father(v_d) = v$
- $Traversed(v) = 0$

Relabeling :
- $Stage(v_d) := Again$

$SNR4$: **Node v_d looks if its has a son not yet included in the "on-building tree"**

Precondition :
- $Stage(v_d) = Again$
- $r \in Sons(v_d)$
- $Stage(r) = A$

Relabeling :
- $Stage(v_d) = W$
- $\mathbf{MTCP\_GRS(G, T; v_d, r)}$

$SNR5$: **Node r informs v_d that it has built the maximal connected tree T_r of $G \setminus v_d$.**

Precondition :
- $Stage(r) = Ended$

Relabeling :
- $Stage(v_d) := Again$
- $\mathbf{SUC(r)[v_d] := true}$

$SNR6$: **Node v_d is informed that its son v is labeled $Ended$**

Precondition :
- $Stage(v_d) = Again$
- $v \in Sons(v_d)$
- $Stage(v) = Ended$

Relabeling :
- $Treated(v_d) := Treated(v_d) \cup \{v\}$
- $Stage(v) := Over$

$SNR7$: **Succorer sons of v_d among the set of sons has been computed**

Precondition :
- $Stage(v_d) = Again$
- $\#Treated(v_d) = \#\, Sons(v_d)$

Relabeling :
- $Stage(v_d) := Over$

$1,$ trivially verified. We suppose that the property is correct for $\#SUCS(v) = l - 1,$ we must prove that the property remains correct for $\#SUCS(v) = l.$ Let u_l be the l^{th} succorer son of $v.$ From the hypothesis, all the succorer sons except u_l may built a spanning tree of their components. We denote by $\mathcal{P}_{l-1}$ the sub-graph induced by the connected components of $SUCS(v) \setminus u_l.$ There are neighbors of v including u_l not in $\mathcal{P}_{l-1}.$ It suffices to consider the graph $G \setminus \mathcal{P}_{l-1},$ so in this graph v has exactly one succorer son $u_l,$ which is like a trivial case. Hence the property is also verified. □

Lemma 7.6.2 *The succorer node procedure requires time bounded by* $t_1 deg(G) \times \#V + t_2$ *time for some constants* $t_1,\ t_2$ *when applied to some node* v_d *of a graph* $G.$

Proof. The cost of the application of the SNP_GRS is the cost of an exploration using a "depth-first trip" on a tree and the cost of the following procedures are applied on the graph $G = (V, E),$ their cost are:

1. $SPPP\_GRS$ and $SPPP\_GRS^{-1}$ are both in $O(\#V),$
2. CP_GRS is in $O(\#V),$
3. $MTCP\_GRS$ is in $O(deg(G) \times \#V),$

Then, the SNP_GRS time complexity is bounded by $t_1 deg(G) \times \#V + t_2$ where $t_1,\ t_2$ are constants. □

Succorer Algorithm ($\mathcal{SUC}$) Now we present an encoding and an example of the *succorer algorithm*. As we shall see, that is an extended version of the algorithm given in the previous Chapter to deal with the test of the 2-vertex connectivity. Here, the investigation will continue in spite of the meet of cut-nodes.

Let $G = (V, E)$ be a graph $G = (V, E)$ and let v_0 be a node in $V_G.$ We denote by (G, L) a labeled graph where $\forall v \in V_G,\ InvStage(v) = N.$ The node v_0 starts the computation: It builds a spanning tree Inv_T of G rooted at v_0 ($SUC1$). Initially, all the vertices of G are such that $InvStage = A.$ Since the test scheme is based on the use of a "depth-first trip" exploration on the tree $Inv\_T,$ the root v_0 is the only one able to start the trip with the rule $SUC2$. Each node ready to be examined switches its label to " On_Test " ($SUC2,\ SUC4$). The examination phase of the node v_d consists on the computation of the set of its succorer sons using the procedure SNP_GRS which we will present in the following. It proceeds as follows. The node v_d labeled " On_Test " does the required initializations of the SNP_GRS using the *configuration procedure* presented shortly below. At the end of this procedure, v_d is labeled $Stage(v_d) = D$ and now v_d is ready to start the computation of its succorer sons ($SUC3$). As we shall see in the following, eventually the set of the succorer sons of v_d will be computed. If v_d is not a leaf node in $Inv\_T,$ it transfers the label "On_Test" to one of its sons. Eventually some node v finishes the test of all the nodes in its sub-tree ($InvSons(v) = \emptyset$). So it informs its father that the computation is over ($SUC5$). Now v is avoided from the list "$InvSons$" of its father which then can continue the exploration choosing one of its not yet tested son. Furthermore, only the root v_0 detects the end of the "trip" when all its sons are over. It means that all the nodes are tested and their succorer sons list are computed. Then, v_0 applies the rule $SUC6$ to affirm the end of the computation of the algorithm.

Algorithm 21 Succorer computation algorithm ($\mathcal{SUC}$)

- **Input:** A graph $G = (V, E)$ and a node $v_0 \in V$.
- **Labels:**
  - $Stage(v)$, $B(v)$
  - $InvStage(v) \in \{A, KO, KO\_ST, N, OK, OK\_ST\ On\_Test, Over, Tested\}$
  - $InvFather(v)$, $InvSons(v)$
- **Initialization:**
  - $\forall v \in V,\ InvStage(v) = N$,
  - $\forall\, v \in V,\ SUC(v) = \emptyset$
- **Result:** A resulted labeling function SUC is a succorer labeling of G.
- **Rules:**

$SUC1$: **Node v_0 starts to build the tree $Inv\_T(InvFather, InvSons)$**

Precondition :
- $InvStage(v_0) = N$

Relabeling :
- $\mathbf{STM\_GRS(G, Inv\_T; v_0; InvStage, N, A; InvFather; InvSons)}$ /*all the nodes v satisfy : $InvStage(v) = A$.*/

$SUC2$: **Node v_0 initializes the test $Inv\_T(InvFather, InvSons)$ /*v_0 is the only node such that $InvFather(v_0) = \perp$ [3]*/**

Precondition :
- $InvStage(v_0) = A$
- $InvFather(v_0) = \perp$
- $InvSons(v_0) \neq \emptyset$

Relabeling :
- $InvStage(v_0) := On\_Test$

$SUC3$: **Node v_d is ready to be tested**

Precondition :
- $InvStage(v_d) = On\_Test$

Relabeling :
- $\mathbf{CP\_GRS(G, Inv\_T; v_d)}$
- $Stage(v_d) := D$
- $\mathbf{SNP\_GRS(G, Inv\_T; v_d)}$
- $InvStage(v_d) := Tested$

$SUC4$: **Node v_d is tested, its "SUC" is computed, one of its son will become the tested node**

Precondition :
- $InvStage(v_d) = Tested$
- $u \in InvSons(v_d)$

Relabeling :
- $InvSon(v_d) := InvSon(v_d) \setminus \{w\}$
- $InvStage(v_d) := W$
- $InvStage(w) := On\_Test$

$SUC5$: **Node v ends the test of its sub-tree**

Precondition :
- $InvStage(v) = Tested$
- $InvSons(v) = \emptyset$
- $InvFather(v) \neq \perp$

Relabeling :
- $InvStage(v) := Over$
- $InvSons(InvFather(v)) := InvSons(InvFather(v)) \setminus \{v\}$
- $InvStage(InvFather(v)) := Tested$

$SUC6$: **Node v_0 detects the end of the succorers computation**

Precondition :
- $InvStage(v) = Tested$
- $InvSons(v) = \emptyset$
- $InvFather(v) = \perp$

Relabeling :
- $InvStage(v) := Over$

7.7 Overall Proofs and Complexities Analysis

In this section, we propose some lemmas and theorems to prove the correctness of the $\mathcal{MOFST}$ algorithm. We start by proving that the labeling SUC computed using the $\mathcal{SUC}$ algorithm succors G. Then, we show that such a labeling is preserved in spite of failures. Finally, we present a short complexity analysis.

Lemma 7.7.1 *The $\mathcal{SUC}$ algorithm presented above computes succorer sons for a graph G.*

Proof. The aim of this lemma is only to prove that the $\mathcal{SUC}$ algorithm computes the succorer sons of all the nodes of the given connected graph G. The first phase of the algorithm uses a STM_GRS procedure, so it computes correctly a spanning tree of G. Since the second phase is based on a "depth-first trip" exploration, it may be easily proved by induction of the number of nodes in G that all the nodes will be reached by such an exploration. □

Then, according to Lemma 7.6.1 and Lemma 7.7.1 we claim the following:

Theorem 7.7.2 *Let $G = (V, E)$ be a connected graph and SUC be the labeling obtained after the execution of the $\mathcal{SUC}$ algorithm. Then, SUC is a succorer labeling of G.*

For the time complexity, the $\mathcal{SUC}$ algorithm is based on the application of the SNP_GRS to all the nodes, except the leaves. That is,

1. The rule $SUC1$ is applied once. Its cost is due to the application of the *spanning tree procedure* to the graph G. So, its cost is in $O(deg(G) \times \#V)$.
2. The rule $SUC2$ is applied once.
3. The *configuration procedure* and the *succorer node procedure* are applied for each node in the investigation tree. Then, the cost of the application of rule $SUC3$ is due to the application of such procedures.
4. The rules $SUC4, SUC5$ and $SUC6$ are used to implement the "depth-first trip" on the tree $Inv\_T = (V, E_T)$. So, the cost of such exploration is $2\#V - 1$ rules.

Then,

Lemma 7.7.3 *The $\mathcal{SUC}$ algorithm requires time bounded by $t_3 deg(G) \times \#V^2 + t_4$ time for some constants t_3, t_4 when applied to a graph $G = (V, E)$.*

Theorem 7.7.4 *Let $G = (V, E)$ be a graph associated with a succorer labeling SUC. If the crash of a node v_f disconnects G into P connected components, then the $\mathcal{MOFST}$ algorithm maintains a forest of spanning trees of the resulting components and updates correctly the succorer sons of any non crashed node.*

Proof. We will prove this lemma by induction on the number of failures denoted by f. For the first case of $f = 1$, the correction follows the identification of the set of maintainer nodes corresponding to the crashed node v_f using the pre-processing task ($\mathcal{SUC}$ algorithm) and the correction of the *maintenance procedure* (see Lemma 6.4.3). We suppose that the algorithm works correctly for $f - 1$ consecutive failures occur. Now, we will show that 's also correct for f failure. To do this it suffices to apply the algorithm on the connected components resulting of the $f - 1$ failures. It is similar to the case of $f = 1$. Thus, the theorem holds. □

Then,

Theorem 7.7.5 *The $\mathcal{MOFST}$ algorithm requires time complexity bounded by $t_5 deg(G) \times \#V + t_6$, for some constants t_5, t_6 when applied on a graph $G = (V, E)$.*

For the space complexity, each node v is labeled $L(v)$ using the two following components:

1. $B(v)$, $Included(v)$, $Terminated(v)$, $Sons(v)$, $Feedbacks(v)$, $To\_Explore(v)$, $Potential(v), To\_Ignore$, $Treated(v)$, $InvSons(v), SUC(v)$.
2. $Stage(v)$, $Father(v)$, $InvStage(v)$, $InvFather(v)$.

Thus, to encode the first component of a label, every node needs to maintain subsets of its neighbors as descendants, for the set of sons for example. So, every node v needs $11deg(v) \times \log deg(v)$ bits to store this component, where $deg(v)$ is the degree of v. By taking into account the other variables used in the second component of labels, we can claim that the space complexity of $\mathcal{MOFST}$ algorithm is in $O(deg(G) \times \log deg(G))$ bits per node. Then,

Theorem 7.7.6 *The $\mathcal{MOFST}$ algorithm requires space bounded by $t_7 + t_8 deg(G) \times \log deg(G)$ bits per node, for some constants t_7, t_8. when applied to a graph G.*

The following result completes the complexity analysis.

Theorem 7.7.7 *Let $G = (V, E)$ be a graph associated with succorer labeling. After any failure, the $\mathcal{MOFST}$ algorithm maintains a forest of spanning trees of the resulting components and updates the succorer sons in $O(deg(G)\#V^2)$ steps and requires space complexity in $O(deg(G) \times \log deg(G))$ bits per node.*

According to the schema used to construct the set of succorer sons, we claim:

Property 7.7.8 *Let G be a graph and let T be a spanning tree of G rooted at r. Let v be any node in the graph admitting the set of nodes $\{u_1 \cdots u_l\}$ as its succorer sons. Then, the maximal trees computed by $r, u_1 \cdots u_l$ are disjoint.*

As stated in the introduction, the following result makes the brdige between the vertex connectivity test and the maintenance of a spanning tree.

Corollary 7.7.9 *Let G be a connected graph and let T be a spanning tree of G rooted at v_0. Then, if G is a 2-connected graph, only v_0 admits one succorer son. The other nodes don't admit any succorer son. It means that to test if some graph G is 2-connected, it suffices to check this property.*

7.8 Maintenance of a Spanning tree For k-Connected Graphs

To design an efficient algorithm we are interested to minimize the number of succorer sons to be updated. So, we would like to do the update of the spanning tree and the succorer labeling of each of the disconnected components in the same phase. Therefore, the application of the $\mathcal{SUC}$ algorithm is either restricted to G as a pre-processing task or does not required.

The main module of the $\mathcal{MOFST}$ algorithm is called *maintenance-update module*. It is invoked after the detection of some crashed node achieved using the *crash detection module*. In this section we present a graph structure of which the protocol presented bellow operates with optimal requirements.

7.8.1 Protocol for 2-Connected Graphs and One Failure

We consider a 2-connected graph in the presence of one failure. Our algorithm starts with a spanning tree of the graph G associated with the network. Such a tree has been obtained as in the previous section. As we will see, this part may be seen as an occurrence of the Menger's theorem as stated in Property 6.3.1.

We consider a graph G with a spanning tree T rooted at v_0. When a node v_f crashes, using the *crash detection module*, the neighbors of v know this crash. In this case when v_f is not the root v_0 this information is propagated to v_0. Then, v_0 is in charge to rebuild a spanning tree of $G \setminus v_f$. In the particular case of a crash of v_0, a pre-chosen son of v_0 called a "succorer son" will assure this charge. So during the construction a spanning tree T of G, the root v_0 chooses locally one of its son as its succorer son. We will use a label Suc to encode such a relation. In T, there is only one node with Suc is equal to $true$. In the sequel, the node in charge of the construction of the new spanning tree will be called the "maintainer node".

Now, we will first present an overview of our algorithm, referred to as $\mathcal{MOST}$ algorithm in the rest of the paper. Then, we describe the three steps of the algorithm in more detail, proving their correctness and analyzing their complexities using assumptions related to the message passing model. Let v_f be a crashed node.

1. *detection of a crash*. Neighbors of the crashed node detect that node v_f is crashed.
2. *propagation*. Father of the crashed node, if it exists, informs the root of the tree that some node in its sons is crashed.
3. *maintenance*. Maintainer updates the spanning tree of $G \setminus v_f$.

The first step is achieved using the *crash detection module* explained above. Step two can be very short if it is the root that has crashed: The succorer son of the root becomes the maintainer node. If v_f is not the root of T, the father of v_f generates an information to be propagated along the simple path linking it to v_0. Then v_0 becomes the maintainer node. Such a task is done using a *simple path propagator module* (see bellow). The third and last step is more delicate. It corresponds to the *maintenance module* detailed in the sequel.

7.8.2 Maintenance of a Spanning tree of k-Connected Graphs and $(k-1)$ Failures

Now we propose an extension of our protocol to deal with k-connected graphs. Thus, we propose a protocol to tolerate at most $k-1$ consecutive failures. That is, if a graph is k-connected, the previous algorithm would be applied iteratively to treat each failure occurrence. As presented above, the algorithm works correctly without any more assumption except the information about the succorer son after each failure occurrence: The succorer son of the actual tree resulting after this crash. So, adding an update of the succorer son of such a tree, the protocol may be extended to deal maintain a spanning tree of k-connected graph tolerating $k-1$ consecutive failures.

To treat each failure occurrence, our distributed algorithm, referred to as $\mathcal{MOST}_k$ algorithm, consists on the following phases:

1. *detection of the crash.* Applying the *crash detection module*.

2. *propagation.* Applying the *simple path propagator module* if the crashed node is not the root.

3. *extended maintenance.* Maintainer node starts the execution of the maintenance update phase.

4. *cleaning*. To deal with the next failure occurrence using the previous three phases, the used labels require some initialization. For example, all the nodes will set their labels $Stage$ to A an so on. It is possible using an extended version of the *marking module* without adding extra complexity.

The extension proposed concerns the *maintenance module*. That is, for a graph G, at each step [4], the $EMSTM$ builds a spanning tree of G deprived of the crashed node. Then the new root checks if it has a succorer son. If the root is newly defined, it needs to choose one of its sons to become its succorer son and it deactivates its status as the succorer son. If the crashed node is a succorer son, then the root chooses, if it's possible, another son to become its succorer son. Otherwise, the maintenance module doesn't update the succorer son. Such a module is denoted $EMSTM$ for extended version of the $MSTM$.

Remark. Our algorithm works correctly for complete graphs tolerating any number of failures, since the graph remains connected.

7.8.3 Implementation

For a 2-connected graph, in the presence of at most one failure the pre-processing is not necessary. Our algorithm works without any more assumption for any node failure except the root. To take into account the particular case of a root failure, we just need to have an extra information: A son of the root has been chosen as its "'succorer son" to start the algorithm. Such a computation may be achieved during the computation of a spanning tree. For a k-connected graph, in the presence of at most $k-1$ consecutive failures, adding to the maintenance procedure, the protocol computes locally the new succorer son in the case of its deletion and in the case of the root deletion. For the other node failure, the update of the succorer son is not required.

To validate such optimizations, we give two possible implementation of this protocol in distributed environment: Using local computations model (GRS) with implicit communications and using message passing model (MPS) with explicit communications. For a given k-connected graph G, where M is the number of its edges, N the number of its nodes and Δ is its degree: In the presence of at most $k-1$ consecutive failures, to deal with each failure occurrence, our GRS algorithm needs the following requirements: $O(\Delta \times N)$ steps and $O(\Delta \log \Delta)$ bits per node. Our MPS algorithm needs the following requirements: $O(M+N)$ messages, $O(N)$ time and $O(\Delta \times \log \Delta)$ bits per node.

7.9 Status and Future Works

This work deals with the problem of maintaining a forest of spanning trees of a graph in a distributed setting using local knowledge. To present easily our contribution, we divided our protocol on a set of phases achieved using a set of procedures. This part of the thesis has three

[4]The step denotes a failure occurrence.

major contributions. First, we present a new formalization of such a problem using the notion of succorer sons. Then, we propose an encoding of a protocol to deal with the maintenance of a forest of spanning tree as an incremental computation of a set of succorer sons and the maintenance of a forest after each failure occurrence. The protocol is encoded in the local computations model. Given a graph $G = (V, E)$ with degree Δ and N nodes, our algorithm requires $O(\Delta \times N^2)$ steps and $O(\Delta \times \log \Delta)$ bits per node. Note, however, that the protocol is applied concurrently to each of the resulted disconnected component. Moreover, this protocol does not need a completely identified network.

The last contribution shows an optimal schedules of our protocol when applied to k-connected graphs. Therefore, a bridge is dressed between our formalization and an occurrence of the Menger's theorem. That is an implementation of its relation with the possible construction of a spanning tree. To validate the last contribution, we propose an encoding of the resulted algorithm in both local computations model and asynchronous message passing model. The algorithm tolerates $k-1$ consecutive failures when applied to a k-connected graph. It's based on an incremental application of the *maintenance procedure*. So at any step, this procedure is responsible for both maintaining of the spanning tree and for updating the succorer son. For a given graph $G = (V, E)$, after each failure occurrence, our algorithm maintains a spanning tree of the resulted graph and computes its corresponding succorer using the following requirements: For the one encoded in the local computations model, it uses $O(\Delta \times N)$ steps and $O(\Delta \times \log \Delta)$ bits per node. For the one implemented in the asynchronous message passing system, it takes $O(N)$ time and uses $O(M + N)$ messages and $O(\Delta \times \log \Delta)$ bits per node.

Now, we propose some further research to extend our works. To improve our solution, rather than to recompute the succorer labeling from scratch each time of the failure occur, it is desirable to include in the *maintenance procedure* the update of the succorer labeling. To design an efficient algorithm we are interested to minimize the number of succorer vectors to be updated. Therefore, the application of the $\mathcal{SUC}$ algorithm is restricted to G as a preprocessing task. Here, we proposed to use graph structure to help us to design an efficient solution. In the future we hope to find solution for arbitrary graphs in the presence of any number of failure.

Our approach assumes that each process is equipped with an unreliable local failure detector, with a minimum required properties to solve a consensus problem [CT96, DGFG+04]. Even, such failure detectors guarantee the reach of "stable" periods: During such periods, the failure detectors are accurate. Thus, our applications work during these periods and such assumption may only make our applications delayed. Another extension is to deal with the case of unreliable failure detectors without assumptions. Thereafter, protocol to maintain spanning tree starts before the reach of the stabilization periods. We are interested to design approximation algorithms to deal with the maintenance problem with errors.

Summary and Further Research

Goals of our work is the study of fault-tolerance in distributed computing based on local knowledge. Our study uses two general models: graph relabeling systems and message passing system. For each of the models, we propose formalization to *express* faults, then the fault-tolerance are encoded as properties to be satisfied by the algorithms. The investigation presented here includes the different mechanisms used in the literature: Self-stabilization, failure detection and local fault-tolerance. Since connectivity is used as a measure of fault-tolerance degree in networks, we studied the problem of vertex connectivity testing. The proposed protocol may be used as a pre-processing task for applications running in unreliable networks. An interface to simulate fault-tolerant distributed algorithms is proposed based on the Visidia software. Several examples of distributed algorithms are given, implemented in Visidia, to validate the proposed approaches. Much attention is associated with some specific problems especially resolving conflicts in the context of self-stabilization, maintenance of spanning tree in the presence of crash failures.

Self-stabilization is a suitable approach to deal with transient failures in distributed computing systems. A system that is designed to be self-stabilizing automatically recovers from an arbitrary state, which is a state reached due, for example, to unexpected failures. Such a property (self-stabilization) is not tied to replications as other well studied fault tolerance techniques and models. We present a formal tool to design and prove self-stabilizing algorithms in the local computations model. A technique to transform an algorithm to a self-stabilizing one is discussed. It is based on two phases. The first phase consists of defining the set of illegitimate configurations (GRSIC). The second phase allows to construct some local correction rules to eliminate the illegitimate configurations. Then the graph relabeling system composed of the initial graph rewriting system improved with the addition of the correction rules is a self-stabilizing system (LSGRS). The transformation is done with a minimum changes since the added correction rules are able to detect and to correct illegitimate configurations and then transient failures locally. The stabilization time of these algorithms is computed in terms of the number of steps or applied rules to reach legal configuration. To measure the "real" stabilization time, we have studied the problem of resolving conflicts. Thus, we propose one possible implementation of the needed synchronizations using local election randomized procedures. That is, the stabilization time is approximated in terms of the required synchronizations.

Unreliable failure detector can be viewed as a distributed oracle that can be asked for information about crashes. Each process has access to its own local detector module which computes the set of processes currently suspected of having crashed. Unreliable failure detectors are characterized by the kinds and the number of mistakes they can make. We have presented a failure detection algorithm for local computations model. The protocol designed has a two-phase procedure : a test phase where processes are tested locally by their neighbors using a

heartbeat strategy (note that the interrogation can be also used), followed by a diffusion phase where test results are exchanged among fault-free processes. Then, the protocol is extended to deal with balls of radius more than 1. The impossibility to implement reliable failure detection protocol in asynchronous distributed systems may be reduced to the impossibility of a consensus problem. Hence, there is no way but to use a time based model. Therefore, we relax the model using a small timing model to implement both an eventually perfect failure detector $\diamondsuit P$ and eventually perfect k-local failure detector $\diamondsuit P_{(f,k)}$. The module is integrated to Visidia in order to design fault-tolerant algorithms whose reliability depends on our failure detector.

As presented along this thesis, that local computations is a high level model to encode distributed computing systems. Here we investigate fault-tolerant distributed algorithms in such a model. Fault tolerance is typically defined according to a particular fault class and a particular tolerance metric. Thus we deal with crash failures as fault class and locality as our tolerance metric. We have presented a method to design fault-tolerant algorithms encoded by local computations. The method consists of specifying a set of illegitimate configurations to describe the faults that can occur during the computation, then adding local correction rules to the corresponding algorithm which is designed in a safe mode. These specific rules are of high priority and are performed in order to eliminate the faults that are detected locally. The resulting algorithm can be implemented under an asynchronous message passing system which notifies the faults using failure detection service for example. It is similar to say that our approach completes the unreliability of failure detection service by the self-stabilization property of the algorithms. We motivate our study by the ability of our designed algorithms to preserve as much as possible some computations that are far from the regions of the faults. Compared with methods based on the initialization of the whole computation after each failure occurrence, our framework uses locality and the initialization concerns only the balls closed to the faults

Especially the latter seems to be an interesting combination of the two flavors of fault-tolerance: Failure detection and self-stabilization which has not been addressed sufficiently in the literature.

In unreliable networks, the possible executions of algorithms increase and the involved processes and messages become larger over a period of time. Thus it is useful to design software to help the designers of such applications to do tests, measurements and then prototyping behind the development. We have presented a powerful method to build an homogeneous interactive visualization of fault-tolerant distributed algorithms based on Visidia software improved by unreliable failure detectors. An interface offers views to select some processes and change their states to simulate their failures. Then, the executions of self-stabilizing and fault-tolerant graph relabeling systems implemented on Visidia show the behavior of these kinds of algorithms. The developed tool helps designers to do some measurements in terms of messages exchanged, to study the complexity analysis, and to understand the possible executions of the distributed algorithms. Furthermore, the tested algorithms can be parametrized through the power of the used failure detection service to achieve an average failure locality better than those studied in the expected case.

The reachability between processes, or the number of paths linking them, is usually used as a metric to measure the fault-tolerance degree of the corresponding network. Since most of the network protocols use graphs to model the network, the metric is in some way the connectivity of the graph. It is useful to design protocols to deal with the test of connectivity

of graphs problem. Indeed, such protocols may be used as a pre-processing task and bricks for algorithms using the connectivity information as input data. Here we deal with the test of the 2-vertex connectivity of graphs in a distributed setting using local knowledge. We present a new formalization of such a problem using the notion of succorer sons: Given a spanning tree of graph $G = (V, E)$, a succorer son of some node v in G is a son which is able to build a spanning tree of $G \backslash v$. Thus, we propose a constructive and a general approach to test whether a given graph is 2-connected. Furthermore, the protocol is encoded in the local computations model and implemented in the message passing model using a set of procedures. The proofs and the complexities analysis for the second implementation are deduced from the first one with simple changes. Thus, a pseudo-bridge is built between the two models. For the k-vertex connectivity test, to our knowledge there is no distributed algorithm based on local knowledge to deal with this problem. Intuitively, we can reduce this problem to the case of $k = 2$ followed by an incremental test procedure.

In addition to the topological properties of the graph modeling the network, some special structures can be used for control purpose in distributed systems. In fact, the use of a spanning tree structure is useful in communication networks because it guarantees the reachability of the network and it minimizes the total communication cost. If such structure is already computed, it is essential to maintain it while crash or disconnection of certain nodes occurs. Here we deal with the problem of maintaining of a forest of spanning trees of a graph in a distributed setting using local knowledge. Using the notion of succorer sons introduced above, we propose a protocol to maintain a forest of spanning trees which is an incremental computations of a set of succorer sons and the maintenance of a forest after each failure occurrence. The protocol is encoded and analyzed in the local computations model. Then, we propose an efficient version of this protocol when applied to k-connected graphs. In this case, our formalization may be seen as an occurrence of the Menger's theorem. The resulted algorithm is implemented in both local computations model and asynchronous message passing model. The algorithm tolerates $k - 1$ consecutive failures when applied to a k-connected graph.

As presented in this thesis, we hope to use simple graph structure to obtain efficient solutions. In fact, our work may be used behind the development of distributed applications in unreliable networks then efficient solutions are easy to be simulated, tested and prototyped rather than complicated solutions in spite of their general application. Now, we propose some further research to extend our works.

Local Computations with Procedures. To give some elegance to the presentation of algorithms presented in our work, we introduce the notion of procedure. Thus, that is opened to us new research field: The use of a procedures or local computations unit (rewriting unit) in the local computations framework as brick to build more complicated algorithms. This allows in first to clarify the encoding of protocols handling a considered number of relabeling rules and labels. What then improve the local computations model without altering its power. The proof of an algorithm using these units is based on the properties of such units. Its complexities measurements are also derived from those of the procedure and the calls. For the use of the procedures to encode distributed algorithms: They simplify their design and their study. But the assumptions about their atomic applications may violate the performances of the designed applications. So it is useful to propose a general methodology for proving their "correctness calls" without such assumptions. This is itself a research topic and the present work is a motivating example.

Graph Relabeling Systems With capabilities. It is essential to introduces a new formalization of computations with degradation information using graph relabeling systems. Each components is associated with a capability to measure and to signify its ability to execute its corresponding task.

We deal with static and dynamic capacity distributions. In this model, the capability is the ability of some process to execute (possibly) incorrectly some rule. Indeed, it is a property about the correctness of the rule application about the states changes. We can consider this property as a probability to change the states. This is a small generalization of the previous model. Thus, this model may be we applied to capture and to simulate failures. An interesting study consists on developing a framework to encode and to prove fault-tolerant distributed algorithms. It is desirable to apply such a tool as a natural and intuitive model to prove impossibility results in distributed computing systems.

Fault-tolerance in Mobile Agents Systems. Mobile agents are programs that are dispatched from a source computer and run among a set of networked computers until they are able to accomplish their expected task. That is, they can migrate autonomously from a computer to a computer. In this system model, faults may happen on the computer, it may also happen during the network communication. Thus, it is important to deal with fault-tolerant mobile agents. Our purpose is to seek models in which errors can be detected and recovered. Further, one of the motivation is the multiple distributed applications based on such a model and the fact that Visidia with mobile agents is now available and also improved by some API to deal with fault handling.

Security. Because of the open and best-effort of the Internet, the used infrastructures like telecommunication systems or grid computing have become increasingly vulnerable both to faults and malicious attacks [PCC97]. This implies that not only fault-tolerance, but also security should be a real concern [EK01].

Avoid all the attacks in the Internet is an illusion. It is also impossible to treat and then to destroy all the vulnerabilities of a system to support the availability and the survivability of distributed systems. That is, some attacks will succeed and then produce "intrusions". So it is necessary to deal with the problem of intrusion tolerance. Indeed, when the intrusion is in some part of the system it doesn't affect its security. We would like to apply the same techniques of fault-tolerance to deal with this problem. The principal differences are: (1) if an attacker succeeds to enter in one part of the system, the difficulties to enter in other parts is not similar. Indeed, the vulnerabilities associated with each part are not necessary the same. (2) It must avoid that a single intrusion in some part of the system informs the attacker about its confidentiality. When the confidentiality is not required, in the sense that it is not considered as a critical information, classical techniques of fault-tolerance may be used: Failure detection and correction and faults masking. Thus, error detection may be based on intrusion detection techniques or on the comparison between diversified executions. Then, the correction consists on the restarting of a system from a correct configuration. Errors masking consists on the availability of the necessary copies of the data and the executions to be able to correct any damage caused by some intrusion. Although, several techniques studied the problem of distributed intrusion detection which may contribute to intrusion tolerance, it is also the preferred target of the attackers. It must be tolerate intrusion itself.

Most works in the literature addresses the combined treatment of both fault-tolerance and

security [MM98]. However, these works do not deal with new issues about a unified treatment of fault-tolerance and security in protocol design. Moreover, the proposed solutions are devoted to some specific infrastructures and do not scale very well to other systems, either because they consider global knowledge (in contrast with the use of local knowledge), or because they do not consider all the possible attacks. Instead the level of fault-tolerance should be comparable to the strength of the security measures, and should therefore be expressed in terms of the security properties. However, the byzantine failure model assumes that an adversary has unlimited computing resources and can alter or modify any messages sent by the processes. Consequently, this would defy any cryptographic security protection, for example. For these reasons, using approaches to specify the fault-tolerance and security open new challenges to their formal verification since security tools are usually considered as black boxes that are offer absolute security. We think that it is useful to seek models to study, to design and to proof distributed algorithms that must be both fault-tolerant and secure.

Simulation and Experimentation. We intend to continue our effort of development of tools establishing our methods and our techniques of validation. Thus, the Visidia tool will be extended to other models, date structures, and techniques to design distributed applications combining abstraction and analyzes methodologies. We also hope to confront our techniques and our tools with significant case studies to help the designers of distributed applications.

Bibliography

[AAE04] P. C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Trans. Program. Lang. Syst.*, 26(1):125–185, 2004.

[AAL87] H. Abu-Amara and M. Loui. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, pages 163–183, 1987.

[Abb90] R. J. Abbott. Resourceful systems for fault tolerance, reliability, and safety. *ACM Comput. Surv.*, 22(1):35–68, 1990.

[ACK90] B. Awerbuch, I. Cidon, and S. Kutten. Optimal maintenance of replicated information. In *31st annual IEEE Symp. on Foundations of Computer Science (FOCS90)*, pages 492–502, October 1990.

[ACK+91] B. Awerbuch, I. Cidon, S. Kutten, Y. Mansour, and D. Peleg. Broadcast with partial knowledge (preliminary version). In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 153–163, New York, NY, USA, 1991. ACM Press.

[ACT02] M.K. Aguilera, W. Chen, and S. Toueg. On the quality of service of failure detector. *IEEE Trans. Comput.*, 51(5):561–580, 2002.

[AD97] Y. Afek and S. Dolev. Local stabilizer. In *ISTCS '97: Proceedings of the Fifth Israel Symposium on the Theory of Computing Systems (ISTCS '97)*, page 74. IEEE Computer Society, 1997.

[ADGFT01] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 108–122, London, UK, 2001. Springer-Verlag.

[ADGFT03] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing omega with weak reliability and synchrony assumptions. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 306–314, New York, NY, USA, 2003. ACM Press.

[AG90] A. Arora and M. Gouda. Distributed reset (extended abstract). In *FST and TC 10: Proceedings of the tenth conference on Foundations of software technology and theoretical computer science*, pages 316–331. Springer-Verlag New York, Inc., 1990.

[AG93] A. Arora and M. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Trans. Softw. Eng.*, 19(11):1015–1027, 1993.

[AH93] E. Anagnostou and V. Hadzilacos. Tolerating transient and permanent failures. In *WDAG93: Distributed Algorithms 7th International Workshop Proceedings*, volume 725 of *Lecture Notes in Computer Science*, pages 174–188. Springer-Verlag, 1993.

[AK00] A. Arora and S.S. Kulkarni. Automating the addition of fault-tolerance. In *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93. Springer-Verlag, 2000.

[AKM+93] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 652–661, New York, NY, USA, 1993. ACM Press.

[AKY97] Y. Afek, S. Kutten, and M. Yung. The local detection paradigm and its applications to self-stabilization. *Theor. Comput. Sci.*, 186(1-2):199–229, 1997.

[AN01] A. Arora and M. Nesterenko. Unifying stabilization and termination in message-passing systems. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 99, Washington, DC, USA, 2001. IEEE Computer Society.

[APSV91] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction (extended abstract). In *Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 268–277. IEEE Computer Society Press, 1991.

[AS96] G. Antonoiu and P. K. Srimani. A self-stabilizing leader election algorithm for tree graphs. *J. Parallel Distrib. Comput.*, 34(2):227–232, 1996.

[AS97] B. Awerbuch and L. J. Schulman. The maintenance of common data in a distributed system. *J. ACM*, 44(1):86–103, 1997.

[AW98] H. Attiya and J. Welch. *Distributed computing, fundamentals, simulations and advanced topics*. Edition McGraw-hill international (UK) limited, 1998.

[BBF01] V. C. Barbosa, M. R. F. Benevides, and A. L. Oliveira Filho. A priority dynamics for generalized drinking philosophers. In *Information Processing Letters*, volume 79, pages 189–195, 2001.

[BBFM99] J. Beauquier, B. Berard, L. Fribourg, and F. Magniette. A new rewrite method for proving convergence of self-stabilizing systems. *In 13th International Symposium on Distributed Computing (DISC), LNCS:1693*, pages 240–253, 1999.

[BDPV99] A. Bui, A. K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing pif in tree networks. In *ICDCS '99: Workshop on Self-stabilizing Systems*, pages 78–85. IEEE Computer Society, 1999.

[BGS94] S. Bhaskar, R. Gupta, and S. A. Smolka. On randomization in sequential and distributed algorithms. *ACM Comput. sur.*, 26(1):7–86, 1994.

[BH02] J. Beauquier and T. Hérault. Fault-local stabilization: The shortest path tree. In *21st Symposium on Reliable Distributed Systems (SRDS 2002), 13-16 October 2002, Osaka, Japan*, pages 62–69. IEEE Computer Society, 2002.

[BHM02] M. Bahramgiri, M.T. Hajiaghayi, and V.S. Mirrokni. Fault-tolerant and 3-dimensional distributed topology control algorithms in wireless multi-hop networks. In *In IEEE Int. Conf. on Computer Communications and Networks (ICCCN02)*, pages 392–397, 2002.

[BHR+00] R. Bosch, P. Hanrahan, M. Rosenblum, G. Stoll, and C.Stolte. Performance analysis and visualization of parallel systems using simos and rivet: A case study. In *Sixth International Symposium on High-Performance Computer Architecture(HPCA)*. IEEE Computer Society, January 2000.

[BHS01] J. Beauquier, T. Herault, and E. Schiller. Easy self-stabilization with an agent. *Proceedings of the 5th International Workshop on Self-Stabilizing systems (WSS'01), Lisbon, Portugal, LNCS:2194*, October 2001.

[BHSZ95] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, and M. Zagha. Nesl user's manual (for nesl version 3.1). Technical report, School of Computer Science, Carnegie Mellon University, 1995.

[BKM97] J. Beauquier and S. Kekkonen-Moneta. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International Journal of System Science*, 28(11):1177–1187, 1997.

[BM03] M. Bauderon and M. Mosbah. A unified framework for designing, implementing and visualizing distributed algorithms. *Electr. Notes Theor. Comput. Sci.*, 72(3), 2003.

[BMMS01] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. Graph relabeling systems : A tool for encoding, proving, studying and visualizing distributed algorithms. *Electronic Notes in Theoretical Computer Science*, 51, 2001.

[BMMS02] M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computation to asynchronous message passing systems. Technical Report RR-1271-02, LaBRI - University of Bordeaux 1, 2002.

[BMS02] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2002)*, pages 354–363, 2002.

[CDPV01] A. Cournier, A.K. Datta, F. Petit, and V. Villain. Self-stabilizing pif algorithms in arbitrary network. *21th International Conference on Distributed Computing Systems (ICDCS 2001)*, pages 91–98, April 2001.

[CDPV02] A. Cournier, A.K. Datta, F. Petit, and V. Villain. Snap-stabilizing pif algorithm in arbitrary networks. *22th International Conference on Distributed Computing Systems (ICDCS 2002)*, pages 199–20, July 2002.

[CGG01] T.D. Chandra, G.S. Goldszmidt, and I. Gupta. On scalable and efficient distributed failure detector. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 170–179. ACM Press, 2001.

[Cha82] EJH. Chang. Echo algorithms: (depth) parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8:391–401, 1982.

[Cha06] J. Chalopin. *Algorithmique distribuée, calculs locaux et homomorphismes de graphes*. PhD thesis, University of Bordeaux 1, 2006.

[CHK02] A. Cherif, N. Hayashibara, and T. Katayama. Failure detectors for large-scale distributed systems. In *21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 404–409, Osaka, Japan, 2002. IEEE Computer Society.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[CS92] M. Choy and A. K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 593–602, New York, NY, USA, 1992. ACM Press.

[CS96] M. Choy and A.K. Singh. Localizing failures in distributed synchronization. *IEEE Trans. Parallel Distrib. Syst.*, 7(7):705–716, 1996.

[CT96] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed system. *Journal of the ACM*, 43(2):225–267, July 1996.

[CW04] W. Chen and K. Wada. Optimal fault-tolerant routings with small routing tables for k-connected graphs. *J. Discrete Algorithms*, 2(4):517–530, 2004.

[Der06] B. Derbel. *Aspects locaux de l'algorithmique distribuée*. PhD thesis, University of Bordeaux 1, 2006.

[DFGO99] X. Défago, P. Felber, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, pages 132–141, Edinburgh, Scotland, September 1999.

[DFP02] M. Duflot, L. Fribourg, and C. Picaronny. Randomized dining philosophers without fairness assumption. In *International Conference on Theoretical Computer Science TCS*, pages 169–180, 2002.

[DGFG$^+$04] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Twenty-Third ACM Symposium on Principles of Distributed Computing (PODC 2004)*, 2004.

[Dij72] E.W. Dijkstra. Hierarchical ordering of sequential processes. In C.A.R. Hoare and R.H. Perrott, editors, *Operating Systems Techniques*. Academic Press, 1972.

[Dij74] E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[DJPV98] A.K. Datta, C. Johnen, F. Petit, and V. Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. In *5th International Colloquium on Structural Information & Communication Complexity, Amalfi, Italy, June 22-24*, pages 229–243, 1998.

[Dol00] S. Dolev. *Self-stabilization*. MIT Press, 2000.

[EH84] A. H. Esfahanian and S.L. Hakimi. On computing the connectivities of graphs and digraphs. *Networks*, pages 355–366, 1984.

[EH91] J. A. Etheridge and M. T. Heath. Visualizing the performance of parallel programs. *IEEE Software*, 08:29–39, September 1991.

[EK01] M. C. Elder and J. C. Knight. Fault tolerant distributed information systems. In *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, page 132, Washington, DC, USA, 2001. IEEE Computer Society.

[ET75] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4(4):507–518, 1975.

[Fel60] W. Feller. *An Introduction to Probability Theory and Its Application, Volume I*. John Wiley and Sons, 1960.

[FG05] F. C. Freiling and S. Ghosh. Code stabilization. In *Self-Stabilizing Systems, 7th International Symposium, SSS 2005, October 26-27, 2005, Proceedings*, volume 3764 of *Lecture Notes in Computer Science*, pages 128–139. Springer, 2005.

[FKTV99] I. Fischer, M. Koch, G. Taentzer, and V. Volle. Distributed graph transformation with application to visual design of distributed systems. *Handbook of graph grammars and computing by graph transformation: vol. 3: concurrency, parallelism, and distribution*, pages 269–340, 1999.

[FLM85] M.J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. In *PODC '85: Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 59–70. ACM Press, 1985.

[FLP85] M.J. Fisher, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FRT01] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Pacific Rim International Symposium on Dependable Computing (PRDC 2001)*, pages 146–153, Seoul, Korea, 2001. IEEE Computer Society.

[GA05] V. K. Garg and A. Agarwal. Distributed maintenance of a spanning tree using labeled tree encoding. In *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*, volume 3648, pages 606–616. Springer, 2005.

[Gab00] H.N. Gabow. Using expander graphs to find vertex connectivity. In *41st Annual Symposium on Foundations of Computer Science*, page 410, 2000.

[Gar99] F. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[GG96] S. Ghosh and A. Gupta. An exercise in fault-containment: self-stabilizing leader election. *Inf. Process. Lett.*, 59(5):281–288, 1996.

[GGHP96] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 45–54. ACM Press, 1996.

[GGT89] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.*, 18(1):30–55, 1989.

[GM91] M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Trans. Comput.*, 40(4):448–458, 1991.

[God02] E. Godard. A self-stabilizing enumeration algorithm. *Inf. Process. Lett.*, 82(6):299–305, 2002.

[Her90] T. Herman. Probabilistic self-stabilization. *Inf. Process. Lett.*, 35(2):63–67, 1990.

[HH04] J. El Haddad and S. Haddad. A fault-contained spanning tree protocol for arbitrary networks. In David A. Bader and Ashfaq A. Khokhar, editors, *ISCA PDCS: Proceedings of the ISCA 17th International Conference on Parallel and Distributed Computing Systems*, pages 410–415. ISCA, September 2004.

[HM] B. Hamid and M. Mosbah. Self-stabilizing applications in distributed environments with transient faults. *International Journal of Applied Mathematics and Computer Science*, Graph Theory and its Applications(to appear).

[HM01] T. Herman and T. Masuzawa. Self-stabilizing agent traversal. In *WSS '01: Proceedings of the 5th International Workshop on Self-Stabilizing Systems*, pages 152–166, London, UK, 2001. Springer-Verlag.

[HM05a] B. Hamid and M. Mosbah. An automatic approach to self-stabilization. In *6th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD2005), Baltimore, USA*, pages 123–128. IEEE Computer Society, May 2005.

[HM05b] B. Hamid and M. Mosbah. An implementation of a failure detector for local computations in graphs. In *Proceedings of the 23rd IASTED International multi-conference on parallel and distributed computing and networks (PDCN'05)*, pages 473–478. ACTA Press, February 2005.

[HM05c] B. Hamid and M. Mosbah. Visualization of self-stabilizing distributed algorithms. In *9th International conference on information visualization IV 2005, London, UK*, pages 550–555. IEEE Computer Society, 2005.

[HM06a] B. Hamid and M. Mosbah. A local enumeration protocol in spite of corrupted data. *JOURNAL OF COMPUTERS (JCP)*, 1(7):9–20, 2006.

[HM06b] B. Hamid and M. Mosbah. A local self-stabilizing enumeration algorithm. In *Distributed Applications and Interoperable Systems, 6th IFIP WG 6.1 International Conference, DAIS 2006*, pages 289–302. Springer, Lecture Notes in Computer Science:4025, June 2006.

[HMRAR98] M. Habib, C. McDiarmid, J. Ramirez-Alfonsin, and B. Reed. *Probabilistic Methods for Algorithmic Discrete Mathematic*. Springer-Verlag, 1998.

[HO94] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. In *SODA: selected papers from the third annual ACM-SIAM symposium on Discrete algorithms*, pages 424–446. Academic Press, Inc., 1994.

[HRG00] M. R. Henzinger, S. Rao, and H. N. Gabow. Computing vertex connectivity: new bounds from old techniques. *J. Algorithms*, 34(2):222–250, 2000.

[HW05a] M. Hutle and J. Widder. On the possibility and the impossibility of message-driven self-stabilizing failure detection. In *Self-Stabilizing Systems, 7th International Symposium, SSS 2005, Barcelona, Spain, October 26-27, 2005, Proceedings*, volume 3764 of *Lecture Notes in Computer Science*, pages 153–170. Springer, 2005.

[HW05b] M. Hutle and J. Widder. Self-stabilizing failure detector algorithms. In *Proceedings of the 23rd IASTED International multi-conference on parallel and distributed computing and networks (PDCN'05)*, February 2005.

[JL99] M. Joseph and Z. Liu. Specification and verification of fault-tolerance, timing, and scheduling. *ACM Transactions on Programming Languages and Systems*, 21(1):46–89, 1999.

[Joh96] B. W. Johnson. An introduction to the design and analysis of fault-tolerant systems. *Fault-tolerant computer system design*, pages 1–87, 1996.

[KNT05] D. J. King, J. H. Nyström, and P. W. Trinder. Are high-level languages suitable for robust telecoms software?. In *Computer Safety, Reliability, and Security, 24th International Conference, SAFECOMP 2005, September 28-30, 2005, Proceedings*, volume 3688 of *Lecture Notes in Computer Science*, pages 275–288. Springer, 2005.

[KP93] S. Katz and K.J. Perry. Self-stabilizing extensions for message-passing systems. *Distrib. Comput.*, 7(1):17–26, 1993.

[KP95] S. Kutten and D. Peleg. Fault-local distributed mending (extended abstract). In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 20–27. ACM Press, 1995.

[KP00] S. Kutten and D. Peleg. Tight fault locality. *SIAM J. Comput.*, 30(1):247–268, 2000.

[KPT03] B. Koldehofe, M. Papatriantafilou, and P. Tsigas. Integrating a simulation-visualisation environment in a basic distributed systems course: a case study using lydian. In *ITiCSE '03: Proceedings of the 8th annual conference on Innovation and technology in computer science education*, pages 35–39, New York, NY, USA, 2003. ACM Press.

[KSV00] D. Kranzlmüller, C. Schaubschläger, and J. Volkert. Array visualization for parallel program debugging. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*. CSREA Press, June 2000.

[KY97] H. Kakugawa and M. Yamashita. Uniform and self-stabilizing token rings allowing unfair daemon. *IEEE Trans. Parallel Distrib. Syst.*, 8(2):154–163, 1997.

[Lam74] L. Lamport. A new solution of dijkstra's concurrent programming problem. In *Commun. ACM*, volume 17(8), pages 453–455, 1974.

[Lap92] J. C. Laprie. *Dependability—Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-tolerant Systems*. Springer-Verlag, 1992. IFIP WG 10.4.

[LM94] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 41–76. Springer-Verlag, 1994.

[LMS99] I. Litovsky, Y. Métivier, and E. Sopena. Graph relabeling systems and distributed algorithms. In World Scientific Publishing, editor, *Handbook of graph grammars and computing by graph transformation*, volume Vol. III, Eds. H. Ehrig, H.J. Kreowski, U. Montanari and G. Rozenberg, pages 1–56, 1999.

[LR81] D. J. Lehmann and M. O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–138, 1981.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[LW93] N. A. Lynch and J. L. Welch. A modular drinking philosophers algorithm. In *Distributed Computing*, volume 6(4), pages 233–244, 1993.

[Lyn96] N. A. Lynch. *Distributed algorithms*. Morgan Kaufmann Publisher, 1996.

[Maz97] A.W. Mazurkiewicz. Distributed enumeration. *Inf. Processing Letters*, 61(5):233–239, 1997.

[MM98] C. Meadows and J. McLean. Security and dependability: Then and now. In *CSDA '98: Proceedings of the Conference on Computer Security, Dependability, and Assurance*, page 166, Washington, DC, USA, 1998. IEEE Computer Society.

[MMR02] A. Mostefaoui, E. Morgaya, and M. Raynal. Asynchronous implementation of failure detectors. *Publication interne n°1484-September*, 2002.

[MMS02] Y. Métivier, M. Mosbah, and A. Sellami. Proving distributed algorithms by graph relabeling systems: Example of tree in networks with processor identities. *In applied Graph Transformations (AGT2002), Grenoble*, April 2002.

[MMWG01] Y. Métivier, M. Mosbah, P. Wacrenier, and S. Gruner. A distributed algorithm for computing a spanning tree in anonymous tprime graph. In *Procedings of the 5th International Conference on Principles of Distributed Systems. OPODIS 2001, Manzanillo, Mexico, OPODIS*, Studia Informatica Universalis, pages 141–158. Suger, Saint-Denis, rue Catulienne, France, 2001.

[MN99] K. Mehlhorn and St. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MOZ05] D. Malkhi, F. Oprea, and L. Zhou. Omega meets paxos: Leader election and stability without eventual timely links. *Distributed Computing: 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005. Proceedings, Lecture Notes in Computer Science*, 3724:339–353, 2005.

[MR95] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, 1995.

[MS] M. Mosbah and A. Sellami. Visidia: A tool for the VIsialization and SImulation of DIstributed Algorithms(2003). http://www.labri.fr/visidia/.

[MSZ00] Y. Métivier, N. Saheb, and A. Zemmari. Randomized rendezvous. In Birkhauser, editor, *Colloquium on mathematics and computer science: algorithms, trees, combinatorics and probabilities*, Trends in mathematics, pages 183–194, 2000.

[MSZ06] M. Mosbah, A. Sellami, and A. Zemmari. Using graph relabeling systems for resolving conflicts. *New Technologies for Distributed Systems (NOTERE'2006), Edition Lavoisier, Hermes*, pages 283–294, June 2006.

[MW87] S. Moran and Y. Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145–151, 1987.

[Nel90] V. P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990.

[Oss05] R. B. Ossamy. *An algorithmic and computational approach to local computations*. PhD thesis, University of Bordeaux 1, 2005.

[PCC97] Critical foundations: Protecting america's infrastructure. Technical report, Report of the President's Commission on Critical Infrastructure Protection, 1997.

[Pet01] F. Petit. Fast self-stabilizing depth-first token circulation. In *Self-Stabilizing Systems, 5th International Workshop, WSS 2001, Lisbon, Portugal, October 1-2, 2001, Proceedings*, volume 2194 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2001.

[PLL97] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the paxos algorithm. In *Lecture Notes in Computer Science: Distributed Algorithms, Proc. of 11th International Workshop, WDAG'97, Saarbrücken, Germany*, volume 1320, pages 111–125. Springer, sep 1997.

[Por99] A. Porat. Maintenance of a spanning tree in dynamic networks. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, page 282. ACM Press, 1999.

[PS04] S. M. Pike and P. A. G. Sivilotti. Dining philosophers with crash locality 1. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 22–29, Washington, DC, USA, 2004. IEEE Computer Society.

[PT98] M. Papatriantafilou and P. Tsigas. Towards a library of distributed algorithms and animations. In *In 4th International Conference on Computer Aided Learning and Instruction in Science and Engineering (CALISCE '98)*, pages 407–410, 1998.

[PV99] F. Petit and V. Villain. Time and space optimality of distributed depth-first token circulation algorithms. In *Distributed Data & Structures 2, Records of the 2nd International Meeting (WDAS 1999), Princeton, USA, May 10-11, 1999: WDAS*, volume 6 of *Proceedings in Informatics*, pages 91–106. Carleton Scientific, 1999.

[PV00] F. Petit and V. Villain. Optimality and self-stabilization in rooted tree networks. *Parallel Processing Letters*, 10(1):3–14, 2000.

[RSW05] M. K. Reiter, A. Samar, and C. Wang. Distributed construction of a fault-tolerant network from a tree. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005),26-28 October 2005, Orlando, FL, USA*, pages 155–165. IEEE Computer Society, 2005.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[Sch93] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, 1993.

[Sel04] A. Sellami. *Des calculs locaux aux algorithmes distribués*. PhD thesis, University of Bordeaux 1, 2004.

[S.G00] S.Ghosh. Agents, distributed algorithms, and stabilization. *COCOON '2000, Springer-Verlag LNCS:1858*, pages 242–251, 2000.

[S.G01] S.Ghosh. Cooperating mobile agents and stabilization. (invited paper). *Proceedings of WSS'01, Springer-Verlag LNCS: 2194*, pages 1–18, 2001.

[SSP85] B. Szymansky, Y. Shi, and N. Prywes. Terminating iterative solutions of simultaneous equations in distributed message passing systems. *In Proceedings of the 4th Symposium on Principles of Distributed computing*, pages 287–292, 1985.

[Sta95] J. T. Stasko. The parade environment for visualizing parallel program executions: A progress report. Technical Report GIT-GVU-95-03, Georgia Institute of Technology, 1995.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*, pages 146–160, 1972.

[Tel00] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, second edition, 2000.

[TS02] A. S. Tanenbaum and M. Steen. *Distributed systems, principles and paradigms*. Prentice-hall, Inc, 2002.

[VÖ4] H. Völzer. A constructive proof for flp. *Inf. Process. Lett.*, 92(2):83–87, 2004.

[Var00] G. Varghese. Self-stabilization by counter flushing. *SIAM Journal on Computing*, 30(2):486–510, 2000.

[Wes01] D. West. *Introduction to graph theory. Second edition.* Prentice-Hall, 2nd ed. edition, 2001.

[YM02] A. Zemmari Y. Métivier, N. Saheb. Randomized local elections. *Information processing letters*, 82:313–320, 2002.