Efficient self-stabilizing algorithms for graphs

Khaled Maamra

université PARIS-SACLAY

NNT : 2017SACLV065

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

Algorithmes auto-stabilisants efficaces pour les graphes

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université de Versailles-Saint-Quentin-en-Yvelines

École doctorale n°580 Sciences et Technologies de l'Information et de la communication (STIC)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Versailles, le 02 octobre 2017, par

Khaled Maâmra

Composition du Jury :

Olivier Bournez Professeur, École Polytechnique.	Directeur de thèse
Shlomi Dolev Professeur, Université Ben-Gurion	Rapporteur
Emmanuel Godard Professeur, Université Aix-Marseille	Examinateur
Mohamed Lamine Lamali Maître de Conférences, Université de Bordeaux	Examinateur
Yannis Manoussakis Professeur, Université Paris-Sud	Président
Laurence Pilard Maître de Conférences, Université de Versailles	Co-directrice
Maria Potop-Butucaru Professeure, Université Pierre et Marie Curie	Rapportrice

Thèse de doctorat

Khaled Maâmra

Efficient Self-Stabilizing Algorithms for Graphs

2017 | Paris, France.

Acknowledgement

'No one can whistle a symphony. It takes a whole orchestra to play it.'
— H E. Luccock

This work would have never been possible without the contribution of many different people. This page is to give back, in ink form, a tiny fraction of what they gave me.

I am very grateful to Olivier and Laurence for giving me the opportunity to do scientific research. They are truly one of the most generous people I have ever met, both scientifically and on a human level. Thank you for always having an open door for me, my questions and my doubts. Thank you for your continuous support, guidance, patience and advice even when things seemed impossible. I learned a lot about research with you. I owe you all, truly.
A special mention goes to Johanne. You have been an amazing co-author and a guiding and experienced voice throughout these years. Together with Laurence, you made working on matchings fun on top of interesting.
The members of the jury took the time to review and evaluate my work. Some of them even had to travel from across the world to attend my defense. I am immensely grateful and honoured.
During this thesis, Prof. Turau gave me the opportunity to visit and work with him in his group at the *Technical University of Hamburg*, Germany. It has been an amazing and productive experience. I can't thank him enough for it.
Results are also the fruit of interesting collaborations, I thank Devan, George, Gerry and Jonas (my academic brother) for being great co-authors and research partners.
Many thanks go to Sandrine, Carole & Hugues for giving me the opportunity to teach and see the other side of research. Many thanks go as well to everyone I taught with, It was an enlightening and enjoyable experience.

Outside research, my PhD years have been the mix of different people and places from *Versailles*, *Polytechnique*, *Hamburg* and Irif. Managing administration across all these institutions wouldn't have been possible without Isabelle, Chantale, Nadia, Fabienne, Mme Kloul, Mme Razik, Mme De Ferron and frau Winterstein. Thank you for keeping up with me and my always not-knowing-that-paper-should-have-been-signed dramas.
The crux of research life are the people you eat, discuss and learn with everyday. You

have been amazing and thanks to you all, getting to work was something I looked forward to. I would like to thank Cyril for being the acolyte of these years, for the encouragement and all the interesting discussions of why Boudebbouz should never play football again. Many thanks to Yoann for all the conversations about economics, introducing me to Sraffa and being a great left back. Tarek for his advice and caring, Ilaria for her good spirit, Michael, Asma, Christina, Amira, Hanane, Bruno (□) for keeping linguistics, sokker.org games and everything else fun, Alex for sleeping in a revolutionary way, Fabian for being the only one who understands what a good font means to the human kind, Thibault for inviting me to lunch always in my native tongue, Laurent, Pablo, Guillaume (Bullet Journal hey !), Finn, Svetlana, Simon, Brieux, Alkida, Denis, Benjamin & Benjamin, Holger, Tobias.

Most importantly, to you close friends, I can't thank you enough for your unconditional support and faith. You probably don't realize it but you have been my family here during these years. Immense love goes to (in no particular order): Amine (Danny !!) for depressing together over Arsenal FC, playing music and getting my mind outside the research bubble. To the Larid family, Karim (e rougi !!) for being such an amazing understanding friend and making eating an everyday challenge, to Barbara for her caring, kindness and for taking me as part of the family to motherly extents at times. To Louiza for making everything else seem futile, being a great kid overall and for inspiring the introductory example of this thesis. To Redouane for his reminders that health is important and his UFC references, to Abdou for the great hikes and being there, to Oussam for understanding exactly how it feels to have mathematics slap you, to Mohammed (El bot) for endless deep discussions, forwardness, sharpness and humour. I love you all.

My last and deepest feelings go to my family. My brother Youcef, since you came everything changed. Thanks for keeping up with me, for the fun we have and all the FIFA beatings. I also can't thank my parents enough, you were always there for me, pushing me and reminding me of what's important. This work wouldn't have been if it wasn't for you. To Amira and Titouhi for being such great siblings and taking care of the family reputation back there in our absence.

I lost two grand parents during this thesis, this work is dedicated to their memory.

Introduction 1

As her fourth year birthday is approaching, Louiza, a popular and smart little girl, is faced with a somehow repetitive phenomenon. She observed that in her last birthday party some lousy chemistry settled in. With some investigations she concluded that this was mainly due to a collective behaviour of her invitee. In fact, everyone during the party tends to speak only to the persons he or she knows. As kids usually act on their own , doing this naively, it often leads to some of them finding themselves with no one to talk to even if they have acquaintances in the crowd. Besides the costly trouble that this engenders, pushing Louiza to a constant swaying between groups of lonely friends, it also has some serious aftermath on the kinder garden cohesion. Luckily, Louiza's father is a mathematician and although he is well-aware that this has an all over dynamical system flavour he is facing a fundamental difference, he can compute the long-run behaviour that Louiza's party will have, but he has no fast means to impose a certain outcome for it. Furthermore, Louiza's friends being a highly hyperactive bunch of kids, any attempt to go through dictating a local behaviour for each kid can be promptly derided since some of them will just sometimes forget to apply the "rules". Louiza's father, hopes to design a minimal set of rules that every kid would apply in order for them to be partitioned into the largest number of couples of talking kids. It also has to take into account the fact that they will from time to time forget to apply the rules for a moment, but hopefully with parents supervision this can never be in an irreversible fashion. So the rules have to ensure that just by applying them the kids would recover from their forgetfulness toward the desired outcome and so without any parental supervision.

Although this problem seems highly artificial, it is not far from arising in real world applications. In the context of the internet for example, a maximum number of computers, just as Louiza's friends, would be required to communicate with other computers known as routers. Every computer in this setting will have to act without any knowledge about the network as a whole, making local decisions to achieve a global goal. Also, we can imagine that instead of forgetting, computers would be exposed to different kind of faults that alter their behaviour and that are due to memory corruptions for example. Given the number of computers on the internet and the asynchronous nature of their interactions it is less artificial in this context to want to achieve the desired task without any human intervention. It is also natural to want the recovery mechanism from these faults to take the least possible amount of time. In the context of computer networks in general such problems are related to the field of distributed systems and self-stabilizing algorithms.

This thesis focuses on designing such algorithms for problems arising in computer network communications. More specifically, when the network is seen as a

mathematical object called a graph it is concerned in designing fast algorithms to partition it (the graph) into the maximum number of connected couples. This is known in graph theory as a matching problem.

This problem has been extensively studied, and found many applications ranging from different abstract areas of mathematics to more practical ones as early as the Second World War. For example, flying two-pilot military aircraft required the two pilots speaking at least one common language. The allies had then, in order to fly the maximum number of aircraft, to partition the set of pilots into connected couples. Connected in this setting is speaking at least one common language. It is just a matching problem and in this thesis it is studied in the context of self-stabilizing algorithms and for any connectivity relation.

To do so, we start by giving some preliminaries in Chapter 2 in which we recall the basic mathematical definitions needed throughout this thesis. These are mainly about graph and set theory. It also serves the purpose of a notational reference.

In Chapter 3, we give the basic computational models from distributed systems, self-stabilization and the tools to compute complexity of algorithms and prove them in these models.

In Chapter 4 we start by tackling the problem of the self-stabilizing maximal matching, that is a less demanding version than requiring a partition into the maximum number of connected couples. Only this is done in an anonymous network, where computers have no mechanism to distinguish each other. The results presented in this chapter have been published in [CLM+16].

In Chapter 5 we try to settle for an approximation of the maximum matching problem in an identified network. We prove that the most recent result concerning this problem has exponential complexity and we give a new polynomial algorithm. This work appears in [CMMP16].

In Chapter 6 we focus on publish/subscribe systems. Where a set of the network computers publishes content and the other subscribes to this content using topics. Every computer that subscribes for a certain topic receives all the publications in the network concerning this topic. We give a self-stabilizing algorithm to route messages from the publishers to the subscribers through the network in an efficient way. It is particularly suited for wireless sensor networks, where limited resource computers don't in many cases have any control on their physical location and therefore on the set of computers they can interact with. The algorithm is implemented and simulated. This chapter then contains the simulation results as well as the general implementation. This work appears in [STM15]

Finally, we conclude this thesis by putting in perspective its different results as well as giving for each of them a discussion on future directions and open problems.

Preliminaries 2

We present, for notational purposes, the basic definitions from set and graph theory as well as the main computational models used throughout this thesis. We assume some familiarity with standard mathematics and propositional logic and when it is too lengthy to do otherwise, we permit ourselves to present some notions informally.

2.1 SETS

The definition of a *set* meets the intuitive notion of a collection of distinct objects, each of which is called an *element*. The easiest way to express a set S is to give the list of symbols representing each of its elements. As an example:

$$S = \{2, 3, 5, 7\}$$

We say that $x \in S$ if x is an element of S and that $x \notin S$ otherwise. A set that does not contain any element is said to be *empty* and is denoted $\varnothing$. On the other hand, the number of elements of a non-empty set S is called the *cardinality* of S and denoted $|S|$. We also allow the cardinality to be infinite, which gives infinite sets as it is the case for $\mathbb{N}$, the set of all natural numbers.
A more concise manner to express a set is to take for its elements only objects that verify a given proposition. This is written as:

$$S = \{x \mid p(x)\}$$

and reads as the set of all elements x for which $p(x)$ holds.

We keep the same notations as in [Lan] for the usual operations on sets:

- $S \cup R = \{x \mid x \in S \vee x \in R\}$ (*Union*)
- $S \cap R = \{x \mid x \in S \wedge x \in R\}$ (*Intersection*)
- $S \setminus R = \{x \mid x \in S \wedge x \notin R\}$ (*Difference*)

When $S \cap R = \varnothing$ we say that S and R are *disjoint*. We extend the arity of the first two operations to a collection of operands $S_0, S_1, \ldots, S_n$ by denoting:

$$\bigcup_{i=0}^{n} S_i = \{x \mid \exists i \in \{0, \ldots, n\} : x \in S_i\}$$

$$\bigcap_{i=0}^{n} S_i = \{x \mid \forall i \in \{0, \ldots, n\} : x \in S_i\}$$

We capture the intuitive idea that a set R is a part of another set S using the notation $R \subseteq S$ defined as:

$$R \subseteq S \equiv (\forall x, x \in R \Rightarrow x \in S)$$

We say, then, that R is a *subset* of S. In addition, if $|R| = k$ we say that R is a *k–element subset* of S. Furthermore, when $R \subseteq S$ and $S \subseteq R$, we say that S and R are *equal*.

2.1.1 *Power sets*

The *power set* of a set S is the set that has for elements all the subsets of S, including the empty set. It is denoted $\mathcal{P}(S)$ and defined as:

$$\mathcal{P}(S) = \{R \mid R \subseteq S\}$$

We also write $\mathcal{P}_k(S), k \in \mathbb{N}$ for the power set of S t hat contains only k–element subsets of S, that is:

$$\mathcal{P}_k(S) = \{R \mid R \subseteq S \wedge |R| = k\}$$

For example, for the set $S = \{1, 2, 3\}$ we have:

$$\mathcal{P}(S) = \{\{1, 2, 3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{1\}, \{2\}, \{3\}, \varnothing\}$$

$$\mathcal{P}_2(S) = \{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$$

$$\mathcal{P}_3(S) = \{\{1, 2, 3\}\}$$

2.1.2 *Partitions*

A set $\{S_0, S_1, \ldots, S_k\}$ of non-empty disjoint subsets of a set S, is called a *partition* of S if :

$$\bigcup_{i=0}^{k} S_i = S$$

2.2 GRAPHS

The main object of study throughout this thesis is *graphs*, and regarding their notation we mainly follow [Die]. A graph is a pair $G = (V, E)$ where V is the set of *vertices* or *nodes* and E a set of 2–element subsets of V representing *edges*. We consider only the case where V is a non-empty finite set. The number of vertices or the *order* of G is $|V|$ and the number of edges is $|E|$. We set $|V| = n$ and $|E| = m$. This definition corresponds to what is known in the literature as *non oriented simple graphs*.

Two vertices u, v are said to be *adjacent* or *neighbours* if $\{u, v\} \in E$. We then refer to the edge $\{u, v\}$ using (u, v) or (v, u) as a shorthand. We also say that (u, v) is *incident* with v and u and that u, v are its *endpoints*. The set of all vertices that are adjacent to a vertex v is called the *neighbourhood* of v and is denoted $N(v)$. Formally, it is defined as follows:

$$N(v) = \{w \in V \mid (v, w) \in E\}$$

The number of elements in $N(v)$ is the degree of v, denoted $d(v) = |N(v)|$. We denote δ the smallest degree in the graph and Δ the largest one. They are expressed as:

$$\delta = \min\{d(v) \mid v \in V\} \quad \Delta = \max\{d(v) \mid v \in V\}$$

One of the advantages of graphs is that they can be easily represented on the plane by drawing a dot for each vertex and a line between every two adjacent vertices. Such a representation is given in figure 2.1, for the following graph

$$G = (\{v, u, w, x, z\}, \{(v, u), (v, x), (v, z), (v, w), (u, w), (u, x), (x, w), (x, z)\})$$

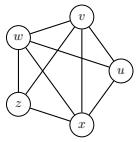


FIGURE 2.1
A representation of a graph. Here $n = 5$, $m = 9$, $d(v) = 4$ and $N(u) = v, w, x$

The reader can easily verify that all the notions defined above can be deduced from the representation of G. For a node v, the degree $d(v)$ is the number of lines attached to the dot v, the neighbourhood $N(v)$ is the set of all dots connected to v with a line. As a result, when speaking about a graph G we refer indistinguishably to the structure G and its representation on the plane.

Subgraphs, induced subgraphs and operations

A graph $G_1 = (V_1, E_1)$ is said to be a *subgraph* of $G_2 = (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. In addition to that, if the following property holds:

$$\forall u, v \in V_1 : (u, v) \in E_2 \Rightarrow (u, v) \in E_1$$

we say that G_1 is an *induced subgraph* of G_2 and that V_1 induces G_2. On the other hand, we say that G_2 *contains* G_1.

Unless specified otherwise, when we talk about a graph G_2 *containing* another graph G_1, it is always in the sense of induced subgraphs. Furthermore, when it is clear from context, we abusively talk about a graph V to refer to the graph $G = (V, E)$ induced by V.

We define the operations of *union* and *intersection* of two graphs G_1, G_2, respectively, as:

- $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$
- $G_1 \cap G_2 = (V_1 \cap V_2, E_1 \cap E_2)$

  When $G_1 \cap G_2 = \varnothing$ we say that G_1 and G_2 are *disjoint* graphs.

Paths, cliques and different families of graphs

DEFINITION 1. *A* path *of* length *k, denoted P^k, is a non-empty graph $P = (V, E)$ on $k + 1$ distinct vertices and in which every two consecutive vertices in V are adjacent. Formally:*

$$V = \{u_0, u_1, \dots, u_k\} \quad E = \{u_0u_1, u_1u_2, \dots, u_{k-1}u_k\}$$

As we did for edges, we adopt a shorthand by referring to a path by giving the list of its vertices $P = u_0u_1, \dots, u_k$. Moreover we say that u_0 and u_k are the endpoints of P^k.

DEFINITION 2. *A* cycle *is a path in which both endpoints coincide, i.e. $u_0 = u_k$.*

DEFINITION 3. *A* complete graph *$K = (V, E)$ is a graph in which every two vertices are adjacent. It is a graph that satisfies $E = \mathcal{P}_2(V)$ and is denoted K^n.*

DEFINITION 4. *A subgraph G_1 of a graph G_2 is said to be* maximal *regarding a property p, if there is no other subgraph in G_2 that contains G_1 and satisfies p.*

Cliques

A *clique* is a maximal complete subgraph.

Connectivity

A graph G is said to be *connected* if for every two vertices u, v in G, there exists a path whose endpoints are u and v. When this does not hold for G we say that G is *disconnected*.

Trees

A *tree* is a a connected graph without cycles.

Distance in graphs

The distance in G between two vertices u, v is the length $dist(u, v)$ of the shortest path (edge wise) in G between u and v. The largest distance in the graph is called the *diameter* of G and denoted $diam(G)$

Matching

A matching M in a graph G is a subset of the edges of G such that no two edges in M share a common endpoint. The largest matching in a graph in terms of cardinality is called a *maximum matching*, and a matching that cannot be extended by adding an edge and still be a matching is said to be *maximal*.

A maximum matching is also a maximal matching.
The matching M is said to be $1-$ *maximal* if no new matching can be produced by removing an edge from M and replacing it by a two other edges from $E \setminus M$.

THEOREM 1. *[HK73]. In a graph G, every* $1-$ *maximal matching is also a* $2/3-$*approximation for the maximum matching*

2.3 ALGORITHMIC SETTING

Graph theory provides a very useful model for different areas of science [Bal85] [MB09]. It is, therefore, very important to be able to compute, in a mechanical way, certain of their properties. This is done through *algorithms*, a set of ordered instructions, or rules, that when followed (we say *executed*) achieve the desired task.

Turing Machines

The aforementioned intuitive definition of an algorithm is very satisfying until we ask the following question:

Can all the properties be computed in such a mechanical fashion ?

A positive answer to this question would consist in giving an algorithm for each of the properties that one wants to compute. On the other hand, in a negative case, one would like to give a proof that no algorithm exists to compute a certain property. Proving this negative case requires a formal definition of what an algorithm is, in order to consider only that definition instead of all the possible algorithms there is. This is done in [Tur36], where Turing introduces the *Turing machine*, an abstract machine that can compute all the properties for which we have an algorithm. Although this latter claim hasn't been proved yet, it is thought to be true following the introduction by Alonzo Church of another definition based on $\lambda-$ *calculus* and showing it to be equivalent to Turing Machines [Chu36]. This led to what is known as the *Church-Turing thesis*:

The intuitive notion of an algorithm is equivalent to the notion of Turing Machines

It is based on the *Church-Turing thesis* that we make no distinction, in this thesis, between a computing entity, an algorithm or a Turing Machine. We, also, won't give a formal definition of a Turing machine, as our results concern rather the interaction of a collection of them. A model of this collective interaction is known as a *distributed system* and is discussed in Chapter 3. The interested reader in Turing Machines is referred to [Sip06] for an introduction to complexity theory.

Algorithms and instances

In the realm of collective behaviours, making no distinction between an algorithm and a computing entity comes with a minor difficulty. One can make a difference between two entities running the same algorithm, whereas this difference doesn't extend when reasoning in terms of algorithms. For this, we introduce the notions of an *instance* and *instantiation* further on in this document.

2.4 CONCLUSION

This chapter served two purposes. The first is to be used as a refresher to simple mathematical notions, the second and the most important is to provide a notational basis for the algorithms that will be written during this thesis. This has been done by giving the used definitions in set theory and also graph theory. Moreover, an rudimentary reflection on complexity theory is introduced.

Distributed Systems and Self-Stabilization

3

This chapter gives the basic definitions and computational models used throughout this document. It also exhibits relevant techniques to either prove results or design algorithms.

3.1 DISTRIBUTED SYSTEMS

Loosely speaking, a *distributed system* is a collection of computing entities interacting through pairwise communication in order to achieve a certain task. Within this collection each entity is responsible only for its own actions, meaning that the interaction between the different entities is not coordinated by any global mechanism (e.g. global time or global view). Moreover, each one of them has a limited view of the system, in the sense that every pair of entities interact only if there exists a communication link between them. As a consequence, initially, no entity is aware of the number of entities in the collection.

Although such a system provides a model for various behaviours, ranging from economics, social sciences to biology [SCWB08] [Kleoo], we consider it only in the case of computer networks. It is in relation to this that we use the term *process* or *node* instead of entity throughout this document.

3.1.1 *Communication graph*

Topologically, a physical network can be viewed as a graph, where vertices are the processes and an edge between two vertices reflects the existence of a communication link between them. Such a graph $G = (V, E)$ is known as a communication graph, where each $v \in V$ represents a process and the value v is said to be its *identifier* or *identity*.

DEFINITION 5. *A communication graph $G = (V, E)$, is a graph where each $v \in V$ represents a process, and every two processes u, v communicate if and only if $(u, v) \in E$.*

A communication graph where it is impossible to distinguish two processes using identities is said to be *anonymous*. If only one process can be distinguished, then the communication graph is said to be *distinguished*, and in the case where all processes can be distinguished we say that the graph is *identified*. Moreover, in a communication

graph G, every process v has a set of special variables known as the *knowledge* of v. These variables can only be read by v and cannot be modified by any process in the graph. Unless specified otherwise, the neighbourhood of a process v is always considered as being part of the knowledge of v.

In addition to the topology, we capture the various actions operated by the different processes using algorithms. Each process runs a local copy of an algorithm $\mathcal{A}$ by duplicating the code of $\mathcal{A}$ and its variables to its local memory. To distinguish this new local algorithm running on a process v from the original one $\mathcal{A}$, as well as from the ones running on other processes, we formalize the notion of an *instance*.

DEFINITION 6. *An instance of an algorithm $\mathcal{A}$ on a process v, denoted $\mathcal{A}(v)$, refers only to the algorithm $\mathcal{A}$ that is running on v.*

Each instance $\mathcal{A}(v)$ is denoted by rewriting $\mathcal{A}$ and subscripting all its variables by the identifier of the process running it. An example of this rewriting is given below for an instance of the Collatz algorithm on a process v. In a computer network, this

Algorithm 1 Collatz algorithm

Require: $x_v \geqslant 0$
1: **while** *true* **do**
2: **if** $x_v \equiv 0 \pmod 2$ **then** ▷ The variable x is rewritten as x_v
3: $x_v \leftarrow x_v/2$
4: **if** $x_v \equiv 1 \pmod 2$ **then**
5: $x_v \leftarrow 3x_v + 1$

rewriting highlights the variables stored locally on v, from the ones stored on other processes. When x_v is such a variable we permit ourselves to write $x_v \in \mathcal{A}(v)$.

An instance of an algorithm is written as a set of rules, with each one expressed as a conditional. This is said to be a *guarded rule* notation with the predicate part of the conditional known as the *guard* and its statements as the *command*. Each statement within the command is known as an *action* and is an assignment of new values to the local variables of v.

if <*guard*> **then** <*command*>

When an algorithm contains more than one rule we name each one of them, to facilitate their referencing in text.

The notions introduced so far are sufficient to define a distributed system.

DEFINITION 7. *A distributed system $\mathfrak{A} = (\mathcal{A}, G)$ is an algorithm $\mathcal{A}$ together with a communication graph $G = (V, E)$ such that there exists a mapping ϕ :*

$$\begin{aligned} \phi : V &\to \mathfrak{A} \\ v &\mapsto \mathcal{A}(v) \end{aligned}$$

where $\mathcal{A}(v)$, is the instantiation of algorithm $\mathcal{A}$ on the node v.

The algorithm $\mathcal{A}$ is said to be a distributed algorithm.

3.1.2 *Communication*

The communication between the different processes of a distributed system varies in nature. For example, two processes u, v linked by a communication link (u, v) can use a set of shared variables to communicate, such that v (resp. u) can read all the shared variables of u (resp. v) but not write to them. Such a communication model is known as a *state model*. Formally:

DEFINITION 8. *A distributed system $\mathfrak{D} = (\mathcal{A}, G)$ is under the state memory model if $\forall v \in V \mid \mathcal{A}(v)$ is such that*

- *The variables $x_u, u \notin N(v)$ do not appear in $\mathcal{A}(v)$*
- *x_u does not appear on the left side of any assignment instruction of the form $x_u \leftarrow y$*
- *The guard of every rule in $\mathcal{A}(v)$ is a predicate exclusively on the variables $\{x_v \in \mathcal{A}(u) \mid v \in V \wedge u \in N(v)\}$*

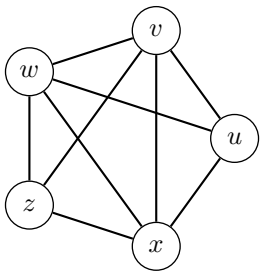


FIGURE 3.1
Example of the interaction in a state model

In the communication graph in Figure 3.1, the node u can read variables of the form x_v, x_x, x_w but not x_z. Moreover, it can write only in ones of the form x_u.

Another way of sharing information is to equip each process with *registers*. These are memory buffers such that a process v can communicate with a process $u \in N(v)$ by writing and reading its register R_{vu} and only reading the register of u, that is R_{uv}. In this case, the variables of v, $x_v \in \mathcal{A}(v)$, are called *intern variables* and cannot be read by u or any other neighbouring process.

DEFINITION 9. *A distributed system $\mathfrak{D} = (\mathcal{A}, G)$ is under the link-register memory model if and only if for every communication link $(u, v) \in E$, there are two registers R_{uv} on u and R_{vu} on v and $\mathcal{A}(v)$ is such that:*

- *The variables $x_u, u \neq v$ do not appear in $\mathcal{A}(v)$*
- *The register R_{uv} does not appear on the left side of any assignment instruction of the form $R_{uv} \leftarrow y$*
- *The guard of every rule in $\mathcal{A}(v)$ is a predicate exclusively on the intern variables of v and R_{vu}, R_{uv} for all $u \in N(v)$*

The models of definition 8 and definition 9 both use as a mean of communication a shared access to each other's memory.

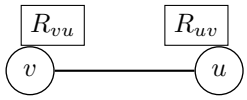


FIGURE 3.2
A couple of nodes communicating through a shared register memory

It is not always possible to implement such a model, especially in cases where processes are geographically far apart. An alternative would be to propagate information through auxiliary processes until it reaches its desired receiver. This is done by exchanging messages using send and receive routines and is known as the *message passing* model. As we cannot always ensure that a message is delivered within a certain interval of time, we consider each communication link as an abstraction of two FIFO queues, one for each process. Messages sent to a process v are stored in its queue as they arrive and are retrieved for treatment from first to last. Moreover each rule in this model is evaluated only upon the reception of a message.

Although the message passing model is the strongest model, the state model will be the main model in which we prove the properties of the different algorithms. This is due to the fact that it is the most abstract model and the reference one to compare the work around our problems.

3.1.3 *Atomicity*

The difference between the models defined above also comes from the *atomicity* assumption. An *atomic step* is the longest portion of code that a process v can execute, such that no change of the state can take place during its run. Such a step is assumed to be instantaneous and is defined as follow:

- In the state model, in one atomic step, a process v can read the local state of all its neighbours and update its whole local state.
- In the link-register model, in one atomic step, a process v can only perform one read or one write on one register and read or write on internal variables for any finite number of times.
- In the message passing model, in one atomic step, a process v can only send or receive a message to/from a neighbouring process but not both at the same time.

For example, it takes a process v under the link-register model $N(v)$ atomic steps to update all its registers, whereas this can be done in one single atomic step under the state model. Therefore, the result of an action in the state model is immediately readable by all its neighbours, which is not the case in the link-register model.

3.1.4 *Description of a distributed system through local states*

To describe the state of a node v within a running distributed system we simply state the values of its memory. This is called the *local state* of v and varies according to the model in which $\mathcal{A}(v)$ is under.

DEFINITION 10. *The local state of $v \in V$ under the state model is the tuple $s_v = (x_v)_{x_v \in \mathcal{A}(v)}$ of the values of all the variables on v.*

DEFINITION 11. *The local state of $v \in V$ under the link-register model is the tuple $s_v = (x_v, R_{vu})_{x_v \in \mathcal{A}(v) u \in N(v)}$ of the values of all the variables on v and its registers.*

DEFINITION 12. *The local state of $v \in V$ under the message passing model is the tuple $s_v = (x_v, q_v)_{x_v \in \mathcal{A}(v)}$ of the values of all the variables on v and the messages contained in its queue.*

From this, we can describe a running distributed system at any given point in time as the set of all local states of all its nodes. This is called a *configuration* of $\mathfrak{D}$.

DEFINITION 13. *A configuration C of a running distributed system $\mathfrak{D}$ is the tuple $C = (s_v)_{v \in V}$. The set of all configurations in G is denoted by $\mathfrak{C}$*

3.1.5 *Execution*

The definition of an atomic step allows us to construct for any two configurations C, C' of the system a sequence $C, A_i, C_{i+1}, A_{i+2}, \dots, C'$ such that $\forall i \in \mathbb{N}^* C_{i+1}$ is obtained by executing a non-empty set A_i of atomic steps. Due to this, we can track the state of the distributed system starting from any initial configuration and give a formal way to characterize what an execution is.

DEFINITION 14. *An execution $\mathcal{E}$ of a distributed system is a sequence*

$$\mathcal{E} = C_0, A_0, C_1, A_1, \dots, C_i, A_i, C_{i+1}, \dots$$

where,

C_0 is a fixed initial configuration and $\forall i \in \mathbb{N}$, C_{i+1} is obtained by executing a non-empty set A_i of atomic steps such that each process has at most one of its atomic actions in A_i.

The sequence $\mathcal{E}$ is maximal, that is, it is either infinite, or finite and no process is activable in the last configuration.

All executions considered in this document are maximal.

The set A_i is said to be a *transition* and if we have C_i, A_i, C_{i+1} in $\mathcal{E}$, we write $C_i \to C_{i+1}$

More generally,

DEFINITION 15. *A configuration C' is said to be reachable from a configuration C if there exist a sequence such that $C, A_0, C_1, A_1, \dots, C_i, A_i, C_{i+1}, \dots C'$. We write $C \rightsquigarrow C'$*

DEFINITION 16. *When the guard of a rule is satisfied in a configuration C we say that the rule is* enabled *in C. Otherwise, it is* disabled.

Similarly for nodes, we define the notion of *activable* or *enabled* in a configuration C

DEFINITION 17. *A node $v \in V$ is said to be activable or enabled in a configuration C if there exists a rule of $\mathcal{A}(v)$ that is enabled in C.*

It is generally assumed that an atomic step in the state model corresponds to the execution of one guarded rule. Therefore, $\forall i \in \mathbb{N}^*$, C_{i+1} is obtained by executing the action of at least one rule that is enabled in C_i. More precisely, A_i is the non empty set of enabled rules in C_i that has been executed to reach C_{i+1} and is such that each process has at most one of its rules in A_i. In the link-register model, the minimal atomicity is not the rule, but the action (remember that a command is a set of actions). Within a command, two types of actions are possible. *Internal actions* are over internal variables, and *communication actions* which consist in reading or writing in a register. An atomic action here is the execution of a finite sequence of internal actions ended by one communication action.

DEFINITION 18. *If in a transition $C_i \rightarrow C_{i+1}$ there is a rule of v in A_i then v is said to be activated.*

3.1.6 *Predicates on executions*

To specify the behaviour of a distributed system we use the notion of a *specification*

DEFINITION 19. *A specification is a predicate on an execution of a distributed system.*

Each problem is associated with a specification that sets the constraints under which the system defines a solution.

Other specifications are used in distributed computing to express the ordering in the interaction between the different processes. *Daemons* are one of them.

3.1.7 *Daemons*

A daemon defines how a system can behave globally regarding the execution of several rules on several nodes during time. This is even important when small differences in the order of execution lead to huge differences in the global state of the system. These orderings often help model the different environments in which a distributed system can evolve, and it can help to give a formal tool to think about them. This is known as a *daemon*. It is a specification that defines for each execution, the set of enabled nodes allowed to move.

We define two properties of daemons, *distribution* and *fairness*.

Distribution

This property ensures a spatial ordering among the different nodes of the network. We use the notion of $k - centrality$ used in [DT11], to express the

predicate that two nodes are allowed to move at the same time only if they are at distance at least k from each other. This is known as a $k-central$ daemon.

DEFINITION 20. *A distributed system $\mathfrak{D} = (\mathcal{A}, G)$ is under the $k-central$ daemon if and only if for every execution $\mathcal{E}$ of $\mathcal{A}$:*

$$\forall i \in \mathbb{N}, \forall u, v \in V : (u \neq v) \wedge (u, v \text{ are activated in } C_i) \Rightarrow dist(u, v) > k$$

In this thesis we will encounter the $0-central$ daemon, known as the *central* or *sequential* daemon and the $diam(G)-central$ daemon, known as the distributed daemon.

PROPOSITION 1. *The set of executions $\mathcal{E}_{k-central}$ of $\mathcal{A}$ under the $k-central$ daemon verify:*

$$\forall k \in \{0, \dots, diam(G) - 1\} : \mathcal{E}_{k+1-central} \subsetneq \mathcal{E}_{k-central}$$

This establishes a hierarchy among the different daemons, according to the distribution property. This is illustrated by Figure 3.3

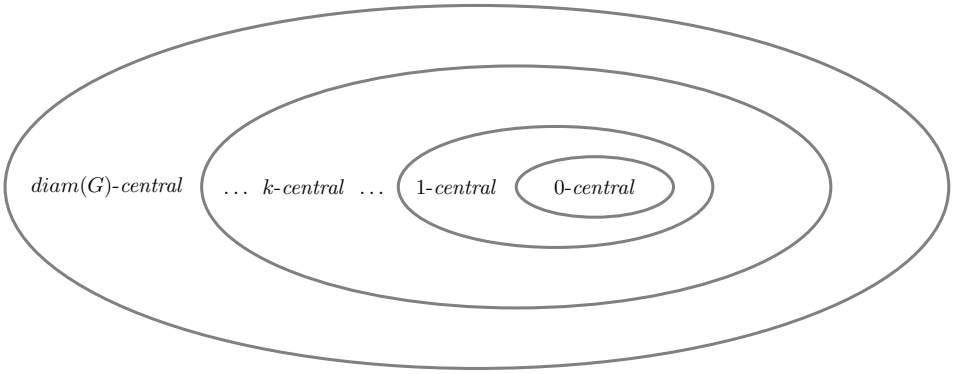


FIGURE 3.3
Hierarchy of daemons according to distribution

The most relaxed daemon is the distributed daemon. In this case, all enabled nodes can be allowed to make a move. This is done at once, concurrently. And once an enabled node makes its move, all the information it used to have about a neighbouring enabled node is obsolete, as the latter moved as well. This makes the coordination efforts between nodes under this daemon very difficult. On the other hand, algorithms executed under the central daemon are easier to design, as only one node moves at the time.

Fairness

The focus in the $k-central$ daemon is global progress, regardless of individual advancement for each node. This can lead to a daemon preventing a continuously enabled process from execution. To exclude executions where such behaviours appear, *weakly fair* daemons were introduced.

DEFINITION 21. *A distributed system $\mathfrak{D} = (\mathcal{A}, G)$ is under the weakly fair daemon if and only if for every execution $\mathcal{E}$ of $\mathcal{A}$:*

$$\neg(\exists i \in \mathbb{N}, \exists v \in V : (\forall j \geqslant i, v \text{ is enabled in } C_j) \wedge (\forall j \geqslant i, v \text{ is not activated in } C_j))$$

Sometimes a stronger version of fairness is needed, one that does not prevent infinitely often enabled processes from execution. This is formalized by the definition below.

DEFINITION 22. *A distributed system $\mathfrak{D} = (\mathcal{A}, G)$ is under the strongly fair daemon if and only if for every execution $\mathcal{E}$ of $\mathcal{A}$:*

$$\neg(\exists i \in \mathbb{N}, \exists v \in V : (\forall j \geqslant i, \exists k \geqslant j, v \text{ is enabled in } C_k) \wedge (\forall j \geqslant i, v \text{ is not activated in } C_j))$$

Daemons without any assumption about fairness are said to be *unfair* or *adversarial*. Those that are weakly fair are simply referred to in this thesis as fair. As with distribution, the fairness property defines a hierarchy among daemons. This is illustrated in Figure 3.4

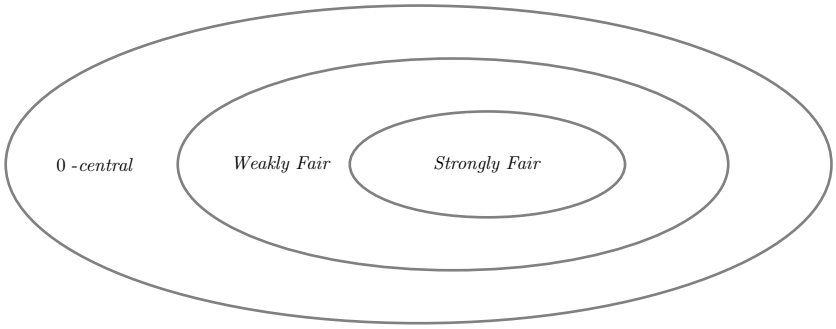


FIGURE 3.4
Hierarchy of daemons according to fairness

A more detailed account of the taxonomy of daemons is given in [DT11].

3.1.8 *Fault-Tolerance*

So far we have always assumed that the instances of the different algorithms of a distributed system are run without interruption, smoothly, toward a pre-set goal. In reality, they depend heavily on their environment of execution. This makes them much more prone to errors of different natures, corrupting memory, communication links, causing crashes and even affecting the code of an entity. Some of these faults are hard to recover from, they are called *Byzantine faults*.

A natural way to classify these faults is by measuring their impact on the system, in other words the duration in which their effect is still observed in the system. This leads to the following categorization:

- *transient faults*, these are any faults that can occur in the system until a certain point in time, beyond which, they never do
- *permanent faults*, these are any faults that always occur in the system beyond a certain point in time.
- *intermittent faults*, are the remaining kind of faults, which can occur at any time in time without restriction.

3.1.9 *Fault-tolerant algorithms*

It is impossible to prevent all types of faults from hitting a distributed system. But we can try to prevent them from making it behave in an undesired way. This is done by tolerating faults and designing the system to recover from them. Such systems are said to be *fault-tolerant*. Two strategies can be used for this purpose, one that handles faults without the observer experiencing any loss of functionality during the recovery process. It is a called a *masking* strategy and requires the recovery mechanism to be permanently running. This can prove costly in different areas of applications. An alternative would be to accept a loss of functionality for the user until the system reaches a good behaviour again. This is known as a *non-masking* strategy.

Two important categories of fault-tolerant algorithms that illustrate these two strategies are:

- *Robust algorithms*, they use a masking strategy by replicating critical algorithms across different entities that take over calculations in case of failures. This type of algorithms is particularly used when only a bounded number of faults can hit the system, since only a finite number of entities can take over. It is also common when there is a high dependency between the computations of the different entities as in scientific computing, like Hadoop or Spark [ZXW+16]. They are often expensive to put in place.
- *Self-stabilizing algorithms* they use a non masking strategy and are designed to recover from any transient faults within a bounded time and without any human intervention. To achieve this, they are designed independently from any initial state.

In this thesis, we design fault-tolerant algorithms to different problems using the latter category of self-stabilizing algorithms.

3.2 SELF-STABILIZATION

In his seminal work [Dij74] Dijsktra introduced the notion of self-stabilizing algorithms as a non masking fault-tolerant solution to the mutual

exclusion problem. This did not gain the expected interest until Leslie Lamport's address [Lam85] during the ACM *Symposium on Principles of Distributed Computing*'83. Today they constitute a sub-field of distributed computing and are part of the standard toolbox of every distributed systems scholar [Telo1][Lyn96] [AW04]. The field reached maturity by the end of the nineties, followed by the publication of a dedicated textbook [Doloo] and the Symposium on Stabilization, Safety, and Security of Distributed Systems.

DEFINITION 23. *A distributed system is said to be self-stabilizing with respect to a specification S, if there exists a subset of legitimate configurations $\mathcal{L} \subseteq \mathfrak{C}$ with the following properties:*

- Correctness*: Every execution that starts from a configuration in $\mathcal{L}$ satisfies the specification S*
- Convergence*: In every execution, a configuration that belongs to $\mathcal{L}$ appears.*

In Figure 3.5 we show an illustration of the executions of a self-stabilizing algorithms.

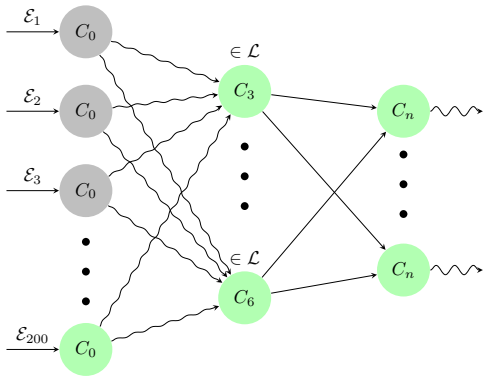


FIGURE 3.5
Executions of a Self-Stabilizing system

The reader can notice that Algorithm 1 is self-stabilizing [1]. It can be started in any initial state, by giving a positive value to the variable x. It has the convergence property as it always reaches a configuration where every node has a value x_v in the trivial cycle $\{1, 2, 4\}$. Also, there is no way in the absence of transient faults to leave this configuration set as any instruction of the algorithm does not change this value for another one outside the cycle. This corresponds to the closure property.

3.2.1 *Expressing self-stabilizing algorithms*

We use the same conventions explained earlier to write self-stabilizing algorithms. Moreover, we always assume that the rules of the algorithm are within

1 This is given only as an example, as this result holds only experimentally. Although it is conjectured to be true, under the name of the Collatz conjecture, it has been neither proved or disproved mathematically

an infinite loop. This is due to the following proposition.

PROPOSITION 2. *A self-stabilizing algorithm does not terminate*

Proof of 2. Suppose that the proposition does not hold. This means that the algorithm terminates at a certain point. Let C_l be the configuration in which the last instance of the algorithm terminates. Every transient fault occurring on the system from C_l and on leads the system to a non legitimate configuration that is impossible to recover from. This violates the convergence property. 2

As we deal only with self-stabilizing algorithms in the remaining of this thesis, we choose to omit the writing of the infinite loop to lighten the notation.

For example, the following algorithm is written in the guarded rule convention. It contains three rules and is self-stabilizing for the maximal matching problem under the central daemon. A proof of this claim as well as a detailed explanation of its rules are given in Section 3.3.

Algorithm 2 Maximal matching algorithm

Require: $N(v)$

▷**Marriage rule**
if $(m_v = \perp) \wedge (\exists u \in N(v) \mid m_u = v)$ **then**
$m_v \leftarrow u$

▷**Seduction rule**
if $(m_v = \perp) \wedge (\forall u \in N(v) \mid m_u \neq v) \wedge (\exists u \in N(v) \mid m_u = \perp)$ **then**
$m_v \leftarrow u$

▷**Abandonment rule**
if $(m_v = u) \wedge (m_u = k) \wedge (k \neq v)$ **then**
$m_v \leftarrow \perp$

A first natural question to ask in this context is whether we can always devise, for any task that can be solved by a Turing Machine, a distributed self-stabilizing algorithm that converges to a solution. It turns out that the answer to this question is positive and that any algorithm that solves a task has a self-stabilizing equivalent that does as well.

THEOREM 2. *[Doloo] A distributed system simulates the universal Turing Machine in a self-stabilizing fashion.*

This result settles the question of decidability for self-stabilizing algorithms. We can focus our interest on the question of efficiency. In the following we introduce the tools used to compute time complexity.

Measuring time

As there is no global timing mechanism in distributed systems, we have to define a measure to evaluate the progress an algorithm makes. This would allow for comparing the efficiency of different algorithms for the same specification / problem.

The most common measure of time is the *move*.

DEFINITION 24. *In a distributed system $\mathfrak{D}$, a node v makes a move if and only if in the execution of $\mathcal{A}$ there exists two configurations C_i, C_{i+1} such that C_i, A_i, C_{i+1} is in $\mathcal{E}$ and v has a rule in A_i.*

When it is relevant, we specify an executed rule by saying that v makes a $<$ *name* − *rule* $>$ move.

Globally, we define

DEFINITION 25. *The number of moves in the execution of $\mathcal{A}$ is $\Sigma_{i=0}|A_i|$*

A move is a node specific measure. A more global one is the *step*.

DEFINITION 26. *A distributed system makes a step each time there is a transition A_i between two configurations C_i, C_{i+1} of the system.*

To measure a balanced global progress among nodes we use *rounds*.

DEFINITION 27. *A round is a minimal portion of the execution in which every node executes at least a move.*

From the definition we can deduce that for every execution of a distributed algorithm:

$$\text{number of rounds} \leqslant \text{number of steps} \leqslant \text{number of moves}$$

3.2.2 *Complexity*

As self-stabilizing algorithms don't terminate it makes more sense to compare them in terms of the time it takes to recover from a transient fault. We consider a worst case analysis, that is done over the execution in which it takes the maximum time to recover from the fault. This time is expressed in terms of the measures seen so far.

DEFINITION 28. *$\mathcal{E}^\ell$ denotes a prefix of an execution $\mathcal{E}$ of a distributed system in which:*

- *No configuration in $\mathcal{E}^\ell$, but the last, is legitimate*
- *the last configuration of $\mathcal{E}^\ell$ is legitimate*

DEFINITION 29. *The worst-case move complexity of a self-stabilizing algorithm $\mathcal{A}$ denotes the maximum number of moves over all $\mathcal{E}^\ell$ of all $\mathcal{E}$ of $\mathcal{A}$ that is:*

$$\max_{\mathcal{E}^\ell}\{\Sigma_{A_i \in \mathcal{E}^\ell}|A_i|\}$$

DEFINITION 30. *The worst-case step complexity of a self-stabilizing algorithm $\mathcal{A}$ denotes the maximum number of steps over the set of prefix-executions $\mathcal{E}^{\ell}$ of $\mathcal{A}$ that is:*

$$\max_{\mathcal{E}^{\ell}}\{k \mid A_k \in \mathcal{E}^{\ell}\}$$

DEFINITION 31. *The worst-case round complexity of a self-stabilizing algorithm $\mathcal{A}$ denotes the maximum number of rounds over the set of prefix executions $\mathcal{E}^{\ell}$ of $\mathcal{A}$.*

As for classical complexity theory, the big-O notation is adopted.

3.2.3 *Other types of self-stabilization*

Ideally, we would want the system, once it reaches a legitimate configuration, to not swing between different legitimate configurations. This could be prevented by reaching a legitimate configuration that doesn't change any further.

DEFINITION 32. *A stable configuration is a configuration where no node is activable.*

DEFINITION 33. *A self-stabilizing algorithm $\mathcal{A}$ is silent, if every execution of $\mathcal{A}$ ends with a stable configuration.*

A weaker form of self-stabilization is the probabilistic one.

DEFINITION 34. *An algorithm $\mathcal{A}$ is self-stabilizing in the probabilistic sense, if it ensures deterministically the correctness property whereas it ensures the convergence only with probability one.*

3.3 PROVING SELF-STABILIZATION

In this section we illustrate a powerful technique for proving the self-stabilization of algorithms. This technique, known as the *potential function* technique is used in this thesis and consists in defining a decreasing (or increasing) function on the set of configurations, such that every move in the system corresponds to a decreasing (or increasing) in the value of the potential function. The idea is to choose a bounded potential function, such that when it reaches its bound, all the configurations having this bound value are legitimate. It is straightforward to deduce the convergence of the system (in the self-stabilizing sense) once this property holds. This is formalized below.

DEFINITION 35. *A function $f : \mathfrak{C} \to \mathbb{K}$ is a decreasing (resp. increasing) potential function on a system $\mathfrak{D}$, if for every execution $\mathcal{E}$ of $\mathcal{A}$ we have:*

$$f(C_{i-1}) > f(C_i) \;\; \forall i \in \mathbb{N}^* \quad (\textit{resp. } f(C_{i-1}) < f(C_i) \;\; \forall i \in \mathbb{N}^*)$$

Where $<$ is any order relation on the set of configurations belonging to $\mathcal{E}$.

To illustrate this technique we prove that Algorithm 2 self-stabilizes into a solution of the maximal matching problem under the central daemon. Every node in this algorithm is equipped with a pointer variable m_v that contains the name of the node that v points to among its neighbours. At any configuration an edge is part of the matching if both its endpoints point the one towards the other. Having $\perp$ as a pointer value means that the node points to no one. Let us recall the specification for this maximal matching problem:

SPECIFICATION 1. *(Specification $\mathcal{M}$)*
For a graph $G = (V, E)$, the set $M = \{(u, v) : m_u = v \wedge m_v = u\}$ is a maximal matching of G, if $\mathcal{M} = \mathcal{M}_1 \wedge \mathcal{M}_2 \wedge \mathcal{M}_3$ holds:

- $\mathcal{M}_1$: $\forall (u, v) \in M : u \in V \wedge v \in V \wedge (u, v) \in E$ *(consistency)*
- $\mathcal{M}_2$: $\forall u, v, w \in V : (u, v) \in M \wedge (u, w) \in M \Rightarrow v = w$ *(matching condition)*
- $\mathcal{M}_3$: $\forall (u, v) \in E, \exists w \in V : (u, w) \in M \vee (v, w) \in M$ *(maximality)*

We define the following states for v:

- **[Matched]** if $m_v = u$ and $m_u = v$
- **[Single]** if $m_v = \perp$ and every neighbour of v is Matched
- **[Waiting]** if $m_v = u$ and $m_u = \perp$
- **[Free]** if $m_v = \perp$ and there is a neighbour that is not Matched
- **[Chaining]** if there exists a neighbour u such that $m_v = u$ and $m_u = k, k \neq v$

Our potential function will have for value at any configuration C:

$$f(C) = (|Matched| + |Single|, |waiting|, |Free|, |Chaining|)$$

Also, the values of f are ordered according to the lexicographic order. For example, $(4, 2, 3, 0)$ is greater than $(4, 2, 2, 1)$.

OBSERVATION 1. *if $f = (n, 0, 0, 0)$ then we are in a stable configuration and this configuration defines a matching. Also for every stable configuration $f = (n, 0, 0, 0)$.*

In such a configuration, all nodes are either matched or single. It is easy to see that this defines a matching as defined in the specification above. Moreover, according to the algorithm, no node can further move from this setting making it a silent convergence.

OBSERVATION 2. *An execution of the Marriage rule reduces the number of free and waiting nodes by 1.*
It increases the number of matched nodes by two.

This leads to an increasing in the value of f.

OBSERVATION 3. *An execution of the Seduction rule reduces the number of free nodes by one and increases the number of waiting nodes by one.*

This leads to an increasing in the value of f.

OBSERVATION 4. *The Abandon rule is executed when v is chaining.*
First, if no neighbouring node points to v then:

- *v becomes free if there exists an unmatched neighbour*
- *v changes to single if all neighbours are matched*

Second case, at least one k points toward v

- *v is changed to free*
- *k is changed from chaining to waiting*

Overall the number of Chaining nodes drops by two, and the number of free and waiting nodes increases by one

This leads to an increasing in the value of f.

THEOREM 3. *Algorithm 2 is self-stabilizing for the maximal matching problem under the central daemon.*

Proof of 3. Every move of the algorithm increases the value of the potential function f. Once the bound $(n, 0, 0, 0)$ is reached we have a legitimate configuration that defines a maximal matching. Moreover, the system does not transition to another configuration as no node makes a further move. Therefore Algorithm 2 is silent. 3

THEOREM 4. *Algorithm 2 stabilizes into a maximal matching in $O(n^3)$ moves.*

Proof of 4. Before reaching the legitimate configuration $(n, 0, 0, 0)$, the system goes, at most, through as many configurations as values of the potential function f. This is roughly $O(n^3)$. The fact that we are in a central daemon allows us to count one move per configuration giving the overall move complexity. 4

A more detailed proof of Algorithm 2 can be found in [HH92]

3.4 DESIGN TECHNIQUES

As with classical algorithms, modularity is an important feature in designing self-stabilizing algorithms. One would like to use and compose already devised solutions as subroutines to solve new tasks. The *Fair Composition*, is a technique that enables us to do this with little to no extra work and still preserve the self-stabilizing property of the whole.

DEFINITION 36. *A fair composition of the distributed algorithms $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ is the set of executions where every node executes a step of each algorithm infinitely often.*

THEOREM 5. *[Doloo] Given the distributed algorithms $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ that are self-stabilizing for the specifications $S_1, S_2, \ldots, S_k$ respectively,*

If no algorithm $\mathcal{A}_i$ changes any of the variables of algorithms $\{\mathcal{A}_j \mid j < i\}$ then the fair composition of $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$ is self-stabilizing for the specification:

$$S_1 \wedge S_2 \wedge \ldots \wedge S_k$$

The idea is that throughout the execution of the composition, $\mathcal{A}_i$ will run regardless of the configuration in which $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_{i-1}$ are in. Once $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_{i-1}$ stabilize, the specifications $S_1, S_2, \ldots, S_{i-1}$ are met and the execution of $\mathcal{A}_i$ will now run from a configuration that verifies the needed specifications. This leads to the stabilization of $\mathcal{A}_i$ to a solution where S_i holds, as well as $S_1, S_2, \ldots, S_{i-1}$.

Having introduced the necessary tools and computational models in this chapter, we follow up by the contributions of this thesis. These are mainly divided in two parts, matching problems and self-stabilizing publish/subscribe systems.

3.5 CONCLUSION

In this chapter we gave a general overview of what it means for a distributed algorithm to be self-stabilizing. We also provided the basic tools to think about them. The state model is the most important of these tools and is what will be mainly used throughout this thesis. We also introduced potential functions. They constitute an important tool to prove self-stabilization. Lastly, we saw how a well defined potential function can be used to prove the self-stabilization of the maximal matching algorithm, therefore introducing the reader to the first graph problem of this thesis and to a classical result in the field of self-stabilizing algorithms.

Maximal matching in anonymous networks

4

In this chapter we provide a self-stabilizing algorithm called AnonyMatch, that is a randomized algorithm for finding a maximal matching in an anonymous network. We show the algorithm stabilizes in $O(n^2)$ moves with high probability under the adversarial distributed daemon. Among all algorithms for adversarial distributed daemons and under the anonymous assumption, our algorithm provides the best complexity as far as we know. Moreover, the previous best known algorithm working under the same daemon and using identities has a $O(m)$ complexity leading to the same order of complexity than our anonymous algorithm. This work appears in [CLM+16]

4.1 INTRODUCTION

Matching problems have received a lot of attention in different areas. In context of dynamic load balancing, job scheduling in parallel and distributed systems can be solved by algorithms using a maximal matching set of communication links [BFM08, GM96] as a basic and black-box function. Moreover, the matching problem has been recently studied in the algorithmic game theory. Indeed, the seminal problem relative to matching introduced by Knuth is the stable marriage problem [Knu76]. This problem can be modelled as a game with economic interactions such as two-sided markets [AGM+11] or as a game with preference relations in a social network [Hoe13]. But, all distributed algorithms proposed in the game theory domain use identities while we are interested in this work in anonymous networks, *i.e.*, without identity. Let us recall from earlier that, a *matching* M in a graph is a set of edges without common vertices. A matching is *maximal* if no proper superset of M is also a matching. A *maximum* matching is a maximal matching with the highest cardinality among all possible maximal matchings. In the following, we present a self-stabilizing algorithm for finding a maximal matching in anonymous networks.

4.2 RELATED WORK

Several self-stabilizing algorithms have been proposed to compute maximal matching in unweighted or weighted general graphs. For an unweighted graph, Hsu and Huang [HH92] gave the first algorithm and proved a bound of $O(n^3)$ on the number of moves under a sequential adversarial daemon. The complexity analysis is completed by Hedetniemi et al. [HJS01] to $O(m)$ moves. Manne et al. [MMPT09] presented a self-stabilizing algorithm for finding a maximal matching. The complexity of this algorithm is proved to be $O(m)$ moves under a distributed adversarial daemon. In a weighted graph, Manne and Mjelde [MM07] presented the first self-stabilizing algorithm for computing a weighted matching of a graph with a 1/2-approximation of the optimal solution. They established that their algorithm stabilizes after at most an exponential number of moves under any adversarial daemon (*i.e.*, sequential or distributed). Turau and Hauck [TH11b] gave a modified version of the previous algorithm that stabilizes after $O(nm)$ moves under any adversarial daemon.

All algorithms presented above, but the Hsu and Huang [HH92], assume nodes have unique identity. The Hsu and Huang's algorithm is the first one working in an anonymous network. This algorithm operates under a sequential daemon (fair or adversarial) in order to achieve symmetry breaking. Indeed, Manne et al. [MMPT09] proved that in an anonymous general network there exists no deterministic self-stabilizing solution to the maximal matching problem under a synchronous daemon. This is a general result that holds whatever the communication and atomicity model (the state model with guarded rule atomicity or the link-register model with read/write atomicity). Goddard et al. [GHJS08] proposed a generalized scheme that can convert any anonymous and deterministic algorithm that stabilizes under an adversarial sequential daemon into a randomized one that stabilizes under a distributed daemon, using only constant extra space and without identity. The expected slowdown is bounded by $O(n^3)$ moves. The composition of these two algorithms can compute a maximal matching in $O(mn^3)$ expected moves in an anonymous network under a distributed daemon.

Self-stabilizing algorithms for optimization problems in anonymous network can sometimes be solved by a deterministic algorithm provided that the latter only needs the distance-2 unique identifier property. This can be achieved by a distance-2 colouring algorithm. A *distance-2 colouring* is a colouring of the graph in which each node has a distinct colour among colours used by any other node within distance 2.

In anonymous networks, Gradinariu and Johnen [GJ01] proposed a self-stabilizing probabilistic algorithm for the distance-2 colouring problem (called *unique naming problem* in this paper. However this scheme requires that every process knows n. They used this algorithm to run the Hsu and Huang's algorithm under an adversarial distributed daemon. However, only a finite stabilization time was proved. Chattopadhyay et al. [CHS02] improved this result by giving a maximal matching algorithm with $O(n)$ expected rounds complexity under the fair distributed daemon. Recall that a *round* is a minimal sequence of moves where each node makes at least one move.

It is straightforward to show that this algorithm stabilizes in $\Omega(n^2)$ expected moves, but Chattopadhyay et al. do not give any upper bound on the move complexity.

In parallel, Emek *et al.* [EPSW14] recently proved every problem that can be solved by a randomized anonymous algorithm, can also be solved by a deterministic anonymous algorithm composed with an underlying randomized distance-2 colouring algorithm. This result is a decidability result and has been extended by a complexity study: Seidel *et al.* [SUW15] established a trade-off between the runtime complexity of the deterministic algorithm and the space complexity of the underlying randomized distance-2 colouring algorithm. However, the result is obtained in a non faulty environment, thus it may not apply in self-stabilizing algorithms.

Figure 4.1 compares features of the aforementioned algorithms and ours. Among all adversarial distributed daemons and with the anonymous assumption, our algorithm provides the best complexity. Moreover, the previous best known algorithm working under the same daemon and using identity has a $O(m)$ complexity leading to the same order of growth as our anonymous algorithm.

Graph	Identified	Anonymous				
Daemon	Adversarial Distributed	Adversarial Sequential	Fair Distributed	Adversarial Distributed		
Reference	[MMPT09]	[HH92, HJS01]	[CHS02]	[GJ01]	Composition of [HH92, HJS01] with [GHJS08]	**Here** AnonyMatch
Complexity	$O(m)$ moves	$O(m)$ moves	$\Omega(n^2)$ expected moves worst case: finite	finite	$O(mn^3)$ expected moves	**$O(n^2)$ moves with high probability**

FIGURE 4.1
Self-Stabilizing Maximal Matching Algorithms

When dealing with matching under anonymous networks, we have to overcome the difficulty that a process has to know if one of its neighbors points to it or to some other node. In Hsu and Huang's paper [HH92], this difficulty is not even mentioned and the assumption a node can know if one of its neighbors points to it is implicitly made. However, this difficulty is mentioned in the Goddard et al. paper [GHS06], where authors present an anonymous self-stabilizing algorithm for finding a 1-maximal matching in trees and rings. To overcome this difficulty, authors assume that every two adjacent nodes share a private register containing an incorruptible link's number. Note that this problem does not appear for the vertex cover problem [TH11a] or the independent set problem [SGH04] even in anonymous networks (see [GK10] for a survey). Indeed, in these kind of problems, we do not try to build a set of edges, but a set of nodes. So, a node does not point to anybody and it simply has to know whether or not one of its neighbours belongs to the set. In the following, we propose a self-stabilizing solution for this problem without assuming any incorruptible memory. Moreover, we show this solution can be applied in all previous anonymous maximal matching algorithms.

4.3 OUTLINE AND MODEL

The AnonyMatch algorithm is the main contribution in this chapter. We also provide other results concerning how to handle the anonymous assumption. In all previous papers dealing with self-stabilizing maximal matching in anonymous systems [CHS02, GHJS08, GJ01, HJS01, HH92], authors make the assumption that nodes can determine whether its neighbour points to itself or to some other node. But, this assumption is not that simple to achieve in anonymous systems since the usual way to do it is using identifiers. In the following, we investigate this question and present a classical algorithm [Dolo00], called LinkName, that gives unique local names to all neighbours of a node in an anonymous system. This algorithm is defined under the link-register model and allows to build a system in which this assumption holds. Then we give a slight modification of AnonyMatch, called AnonyMatch2, leading to a solution that does not need this assumption anymore. AnonyMatch is defined under the state model while its modified version has to be specified under the link-register model. We finally show that this method can be used in all previous anonymous matching algorithms, leading to the following conclusion: the assumption that a node can know if a neighbour points to itself or to some other node in an anonymous system is meaningful.

4.4 THE MAXIMAL MATCHING ALGORITHM AnonyMatch

The *matching algorithm* AnonyMatch presented in this section uses the state model given earlier and is based on the maximal matching algorithm given by Manne et al. [MMPT09]. We start with the description of AnonyMatch and then we will compare it with the Manne et al. algorithm.

In the algorithm AnonyMatch, every node u has one local variable β_u representing the node u is matched with. If u is not matched, then β_u is equal to $\perp$. Algorithm AnonyMatch ensures that a maximal matching is eventually built.

Formally, we require the following specification $\mathcal{M}$:

SPECIFICATION 2. *(Specification $\mathcal{M}$)*
For a graph $G = (V, E)$, the set $M = \{(u, v) : \beta_u{=}v \wedge \beta_v{=}u\}$ is a maximal matching of G, i.e. *if $\mathcal{M} = \mathcal{M}_1 \wedge \mathcal{M}_2 \wedge \mathcal{M}_3$ holds:*

- $\mathcal{M}_1$: $\forall (u, v) \in M : u \in V \wedge v \in V \wedge (u, v) \in E$ *(consistency)*
- $\mathcal{M}_2$: $\forall u, v, w \in V : (u, v) \in M \wedge (u, w) \in M \Rightarrow v = w$ *(matching condition)*
- $\mathcal{M}_3$: $\forall (u, v) \in E, \exists w \in V : (u, w) \in M \vee (v, w) \in M$ *(maximality)*

For the sake of simplicity, we assume that if any node u having the value v in its β_u variable such that $v \notin V$ or $v \notin N(u)$ then u understands this value as null ($\perp$).

Algorithm AnonyMatch has the three rules described in the following. If node u points to null, while one of its neighbors points to u, then u accepts the proposition,

meaning u points back to this neighbor (*Marriage* rule). If node u points to one of its neighbors while this neighbor is pointing to a third node, then u abandons, meaning u resets its pointer to null (*Abandonment* rule). If node u points to null, while none of its neighbors points to u, then u searches for a neighbor pointing to null. If such a neighbor v exists, then u points to it (*Seduction* rule). This seduction can lead to either a marriage between u and v, if v chooses to point back to u (v will then execute the *Marriage* rule), or to an abandonment if v finally decides to get married to another node than u (u will then execute the *Abandonment* rule).

DEFINITION 37. *We define the probabilistic function choose(X) that uniformly chooses an element in a finite set X.*

Algorithm 3 $\mathcal{A}$NONY$\mathcal{M}$ATCH - Maximal matching algorithm in anonymous networks

Require: $N(v)$

▷**Marriage rule**
if $(\beta_u = \perp) \wedge (\exists v \in N(u) : \beta_v = u)$ **then**
  $\beta_u \leftarrow v$

▷**Seduction rule**
if $(\beta_u = \perp) \wedge (\forall v \in N(u) : \beta_v \neq u) \wedge (\exists v \in N(u) : \beta_v = \perp$ **then**
  if *choose*$(\{0, 1\}) = 1$ **then**
    $\beta_u \leftarrow$ *choose*$(\{v \in N(u) : \beta_v = \perp\})$
  else
    $\beta_u \leftarrow \perp$

▷**Abandonment rule**
if $(\exists v \in N(u) : \beta_u = v \wedge \beta_v \neq u \wedge \beta_v \neq \perp)$ **then**
  $\beta_u \leftarrow \perp$

The node that u chooses to get married with in the marriage rule is not specified, since this choice has no bearing upon the correctness nor the complexity of the algorithm.

$\mathcal{A}$NONY$\mathcal{M}$ATCH is a transformation of Manne et al. algorithm that allows to obtain an algorithm defined under an anonymous system from an algorithm that needs unique identifiers in the system. Since the daemon we consider is the adversarial distributed one, the algorithm needs to be probabilistic [MMPT09]. This transformation consists in the suppression of the identifier's comparaisons in the Seduction and Abandonment rules, the suppression of the Update rule and finally the addition of a coin flip action in the Seduction rule. With this coin flip, a node that was able to apply the Seduction rule in the Manne et al. algorithm will seduce only with probability 1/2 in $\mathcal{A}$NONY$\mathcal{M}$ATCH.

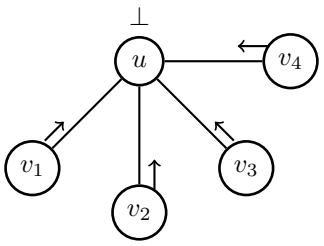
The proof of this algorithm is based on a potential function. To define this

function, we first need to define notions of a *Single node*, a *good edge* and an *almost good edge*.

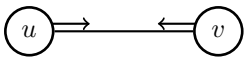
Let C be the set of all possible configurations of the algorithm. Let $C \in \mathcal{C}$ be a configuration. A node pointing to null and having no neighbour pointing to it is called a *Single node*. Two nodes pointing one toward the other are called *Married nodes*. A node pointing to null and toward which other nodes point is called *Undecided*. We define the predicates:

- *Single(u)* $\equiv [\beta_u = \perp \wedge (\forall v \in N(u) : \beta_v \neq u)]$.
- *Undecided(u)* $\equiv [\beta_u = \perp \wedge (\exists v \in N(u) : \beta_v = u)]$.
- *Married(u)* $\equiv [\exists v \in N(u) : \beta_u = v \wedge \beta_v = u]$.

Moreover, we define the set $S(C)$ (*resp.* $U(C)$, $M(C)$) as the set of Single (*resp.* Undecided, Married) nodes in C.



(a) u is an undecided node.



(b) Two married nodes.

FIGURE 4.2
Two Married nodes and an Undecided node. Variable β is represented by the arrows

We now introduce the function $f : C \in \mathcal{C} \mapsto \mathcal{M}(C) \cup \mathcal{U}(C)$. In the following, we prove that f is a potential function for the maximal matching.

LEMMA 1. *Let* $\mathcal{E} = C_0, A_0, \ldots, C_i, A_i, C_{i+1}, \ldots$ *be an execution of* AnonyMatch. *We have:*

$$\forall i \in \mathbb{N}, f(C_i) \subseteq f(C_{i+1})$$

Proof of 1. Consider a configuration C_i and $u \in f(C_i)$ a Married or Undecided node in this configuration.

If $u \in \mathcal{M}(C_i)$, $\beta_u = v \in N(u)$, and u cannot execute rule Marriage or rule Seduction. Since, by definition of $\mathcal{M}(C_i)$, we also have $\beta_v = u$, it cannot either execute rule Abandonment. Thus, u executes no rule. Now, v is also Married, and executes no rule: thus, after A_i, we still have $\beta_u = v$ and $\beta_v = u$ in C_{i+1}, so that $u \in \mathcal{M}(C_{i+1})$.

If $u \in \mathcal{U}(C_i)$, then $\beta_u = \perp$. Thus, it cannot execute rule Abandonment. By definition of $\mathcal{U}(C_i)$, there exists v in $N(u)$ such that $\beta_v = u$: u cannot execute rule Seduction either. Now, consider any node $v \in N(u)$ such that $\beta_v = u$: the guards

forbid it to execute the Marriage or Seduction rules (because $\beta_v \neq \perp$) and the Abandonment rule (because $\beta_v = u$ and $\beta_u = \perp$). Thus, v takes no action, and we still have $\beta_v = u$ in C_{i+1}. Now, if u executes the Marriage rule, then in configuration C_{i+1}, $\beta_u = v$ and $\beta_v = u$, thus $u \in \mathcal{M}(C_{i+1})$. Finally, if u executes no rule, then $\beta_u = \perp$ and $\beta_v = u$. So, $u \in \mathcal{U}(C_{i+1})$.

Thus, a Married or Undecided node in configuration C_i remains Married or Undecided in configuration C_{i+1}: f is non-decreasing in any execution.

□1

Now, the following lemma establishes that in any step in which a Single node is activated, f increases with probability $\geqslant \frac{1}{4}$.

LEMMA 2. *Let $\mathcal{E} = C_0, A_0, \ldots, C_i, A_i, C_{i+1}, \ldots$ be an execution of AnonyMatch. For all $i \geqslant 0$, if A_i contains a move of a node that is Single in C_i, then $f(C_i) \subsetneq f(C_{i+1})$ with probability greater than $\frac{1}{4}$. More formally:*

$$\forall i \in \mathbb{N}, Pr[f(C_i) \subsetneq f(C_{i+1}) | \exists u \in S(C_i) : u \text{ is activated in } C_i] \geqslant \frac{1}{4}$$

Proof of 2. Consider a configuration C_i and let u be a Single node activated in C_i. Let $F(u) = \{v \in N(u) : \beta_v = \perp\}$ and d be its cardinality. Since u is activated in C_i, $F(u)$ is not empty and $d \geqslant 1$.

With probability $\frac{1}{2}$, u decides to seduce a node $v \in N(u)$ such that $\beta_v = \perp$. Then, there are three cases (see Figure 4.3):

- **Case 1)** $v \notin S(C_i)$ **and** v **is activated in** C_i**:** Because $v \notin S(C_i)$ and $\beta_v = \perp$, there exists a node $w \in N(v)$ such that $\beta_w = v$: $v \in \mathcal{U}(C_i) \subseteq f(C_i)$. On the opposite, w cannot be in $f(C_i)$. The only rule that v can apply is the Marriage rule. After applying it, $v \in \mathcal{M}(C_{i+1}) \subseteq f(C_{i+1})$ and $w \in \mathcal{M}(C_{i+1}) \subseteq f(C_{i+1})$

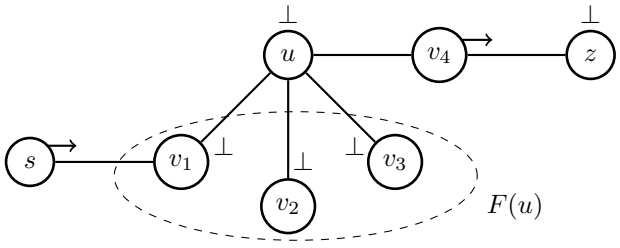


FIGURE 4.3
Example of a configuration where u is a Single node and $F(u) = \{v_1, v_2, v_3\}$.

- **Case 2)** $v \in S(C_i)$ **and** v **is activated in** C_i**:** In this case, process $v \notin f(C_i)$ can only apply the Seduction rule and still points to null with probability $\frac{1}{2}$ in configuration C_{i+1}. Thus, $v \in \mathcal{U}(C_{i+1}) \subseteq f(C_{i+1})$ with probability $\frac{1}{2}$ (moreover, v chooses to point to u with a probability > 0, in which case, $v \in \mathcal{M}(C_{i+1}) \subseteq f(C_{i+1})$).
- **Case 3)** v **is not activated in** C_i**:** In this case v still points to null in C_{i+1}. Thus, $u \in \mathcal{U}(C_{i+1}) \subseteq f(C_{i+1})$.

(Case 1 is illustrated by v_1 and Case 2 by v_2 and v_3 in Fig. 4.3.)

In all three cases, $f(C_{i+1}) \subsetneq f(C_i)$ with probability $\geqslant \frac{1}{2}$. Thus, if a Single node u is activated, then it tries to seduce a node with probability $\frac{1}{2}$, and whatever the status of this node, with probability $\geqslant \frac{1}{2}$, this leads to a new node in $f(C_{i+1})$. Thus with probability $\geqslant \frac{1}{4}$, when a Single node is activated, f increases.

2

LEMMA 3. *Let $\mathcal{E} = C_0, A_0, \ldots, C_i, A_i, C_{i+1}, \ldots$ be an execution of AnonyMatch. For all $i \geqslant 0$, if A_i contains a move of a node that is Undecided in C_i, then $f(C_i) \subsetneq f(C_{i+1})$.*

Proof of 3. Consider a configuration C_i such that an Undecided node u is activated in A_i. Then, u is such that $\beta_u = \perp$ and $\exists v \in N(u) : \beta_v = u$. u can only apply the Marriage rule and set $\beta_u := v$. v is such that $\beta_v = u$ and $\beta_u = \perp$. This implies that $v \notin f(C_i)$, and v cannot be activated in this step. Thus, in C_{i+1}, we have $\beta_u = v$ and $\beta_v = u$. Finally, $v \in f(C_{i+1})$, and f increases. 3

LEMMA 4. *In any execution containing $n + 1$ moves, at least one Undecided or Single node is activated.*

Proof of 4. The seduction rule can only be executed by Single nodes and the marriage rule can only be executed by Undecided nodes. If the Abandonment rule is executed by some node u, then u becomes Single or Undecided and all Undecided, respectively Single, neighbours of u remain so. Thus, after n applications of the Abandonment rule, all nodes are either Single or Undecided, and the next action will activate a Single or Undecided node. 4

This establishes an upper bound of the number of moves between two activations of Undecided or Single nodes. From the combination of Lemmas 1, 2 and 4, we obtain Theorem 6.

THEOREM 6. *Under the adversarial distributed daemon and with the guarded-rule atomicity, the matching algorithm AnonyMatch is self-stabilizing and silent for the specification $\mathcal{M}$ and it reaches a stable configuration in $O(n^2)$ expected moves.*

Proof of 6. First, we prove that a stable configuration satisfies specification $\mathcal{M}$. Let C a stable configuration. By stability of C, for any node u, all guards are false.

Let $(u, v) \in M$. By definition, $v = \beta_u$, so that $(u, v) \in E$: $\mathcal{M}_1$ is true.

Let u, v and w three nodes, with $(u, v) \in M$ and $(u, w) \in M$. By definition, we have $\beta_u = v$ and $\beta_u = w$, so that $v = w$: $\mathcal{M}_2$ holds.

Finally, consider $(u, v) \in E$. $\beta_u \neq \perp$, or $\beta_v \neq \perp$: indeed, if $\beta_u = \beta_v = \perp$, either $\exists x \in N(u) : \beta_x = u$, and u can apply the Marriage rule, or $\forall x \in N(u) : \beta_x \neq u$, in which case it can apply the Seduction rule. Without loss of generality, suppose that $\beta_u \neq \perp$, and note $w = \beta_u$. Then, for u not to be in position to execute the Abandonment rule, we necessary have: $\beta_w = u \vee \beta_w = \perp$. Now, β_w cannot be $\perp$, because w could then apply the Marriage rule. So, $\beta_w = u$, and $(u, w) \in M$, which proves $\mathcal{M}_3$.

Thus, a stable configuration satisfies the specification.

Now, let us prove that a stable configuration is reached in an expected $O(n^2)$ moves. $f(C)$ being a set of nodes, it can increase at most n times in an execution.

Now, each time a Single or Undecided node is activated, f increases with probability $\geqslant \frac{1}{4}$ (after Lemma 2 and 3). By independence of these attempts, f increases in an expected 4 activations of a Single or Undecided node. Now, in every $n+1$ steps, at least one step activates a Single or Undecided node. Thus, in less than an expected $4n+4$ steps, f increases.

Thus, in an expected $4n^2+4n$ steps, f increases n times, and so, there cannot be more than an expected $O(n^2)$ steps before reaching a stable configuration.

□6

THEOREM 7. *Let $0 < p < 1$, and take $k \geqslant \max\{2n^2-2, -8(n+1)\ln p\}$. Then, after k moves, the algorithm AnonyMatch has converged with probability greater than $1-p$.*

Proof of 7. Let F_k be the random variable that is the cardinality of the value of f after k moves. The algorithm has converged before or when $F_k = n$.

Let X_i be the random variable that is 1 if the ith activation of an Undecided or Single node increments the cardinality of f, and 0 otherwise. The X_i variables are independent: the success of a node activation does not depend on the success of the previous activation.

By Lemma 4, at least an Undecided or Single node is activated in any $n+1$ moves, so that the first ℓ activations of such nodes occur in the first $(n+1)\ell$ steps and we have: $F_{(n+1)\ell} \geqslant \sum_{i=1}^{\ell} X_i$. Thus,

$$Pr[F_{(n+1)\ell} \leqslant n-1] \leqslant Pr\,[\sum_{i=1}^{\ell} X_i \leqslant n-1]$$

Moreover, the expectation of any X_i is higher than $\frac{1}{4}$, and so is the expectation of $\frac{1}{\ell}\sum_{i=1}^{\ell} X_i$.
Thus,

$$\begin{aligned} Pr\,[\frac{1}{\ell}\sum_{i=1}^{\ell} X_i \leqslant \frac{n-1}{\ell}] &= Pr\,[\frac{1}{\ell}\sum_{i=1}^{\ell} X_i - \frac{1}{4} \leqslant \frac{n-1}{\ell} - \frac{1}{4}] \\ &\leqslant Pr\,[\frac{1}{\ell}\sum_{i=1}^{\ell} X_i - \mathbb{E}\,[\frac{1}{\ell}\sum_{i=1}^{\ell} X_i] \leqslant \frac{n-1}{\ell} - \frac{1}{4}] \end{aligned}$$

Now, by Hoeffding's inequality applied to the Bernoulli variables $X_1, X_2, \ldots, X_\ell$:

$$Pr\,[\frac{1}{\ell}\sum_{i=1}^{\ell} X_i - \mathbb{E}\,[\frac{1}{\ell}\sum_{i=1}^{\ell} X_i] \leqslant \frac{n-1}{\ell} - \frac{1}{4}] \leqslant \exp\,(-2\ell\,(\frac{n-1}{\ell} - \frac{1}{4})^2)$$

which gives:

$$Pr\,[\frac{1}{\ell}\sum_{i=1}^{\ell} X_i \leqslant \frac{n-1}{\ell}] \leqslant \exp\,(-2\ell\,(\frac{n-1}{\ell} - \frac{1}{4})^2)$$

Taking $\ell \geqslant \max\{2n-2, -8\ln p\}$, we have $\frac{n-1}{\ell} \leqslant \frac{1}{2}$, so that $|\frac{n-1}{\ell} - \frac{1}{4}| \leqslant \frac{1}{4}$, and $(\frac{n-1}{\ell} - \frac{1}{4})^2 \leqslant \frac{1}{16}$; we also have $-2\ell \leqslant 16\ln p$, so that $-2\ell\,(\frac{n-1}{\ell} - \frac{1}{4})^2 \leqslant \ln p$.

Thus, for $\ell \geqslant \max\{2n-2, -8\ln p\}$,

$$Pr\,[\frac{1}{\ell}\sum_{i=1}^{\ell} X_i \leqslant \frac{n-1}{\ell}] \leqslant p$$

Now, for $k \geqslant \max\{2n^2-2, -8(n+1)\ln p\}$, we obtain, by setting $\ell = \frac{k}{n+1} \geqslant \max\{2n-2, -8\ln p\}$:

$$Pr[F_k \leqslant n-1] \leqslant Pr\,[\frac{1}{\ell}\sum_{i=1}^{\ell} X_i \leqslant \frac{n-1}{\ell}] \leqslant p \tag{1}$$

As the Algorithm AnonyMatch has converged at a step k before $F_k = n$, it has converged after $\max\{2n^2-2, -8(n+1)\ln p\}$ moves with probability greater than $1-p$.

□ 7

Using $p = \frac{1}{n}$ in Theorem 7 yields that after $\max\{2n^2-2, 8(n+1)\ln n\} = O(n^2)$ moves, the algorithm has converged with a probability greater than $1 - \frac{1}{n}$. We can conclude:

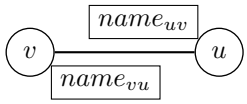
COROLLARY 1. *Algorithm AnonyMatch converges in $O(n^2)$ moves with high probability.*

4.5 HANDLING THE ANONYMOUS ASSUMPTION

Recall from earlier that when dealing with matching under anonymous networks, we have to overcome the difficulty that a node has to know if one of its neighbours points to it or to some other node. In the marriage rule for example, a node u tests if there exists one of its neighbours v such that $\beta_v = u$. Thus u has to know that the value β_v represents itself while u has no identity. This is a fundamental difficulty that is inherently associated to the problem specification when assuming at the same time an anonymous system and the state model . We propose to solve this issue by adopting the *link-register* model.

In this section, we present a classical self-stabilizing algorithm [Doloo], called LinkName algorithm, that allows a node to give unique names to each of its incident communication links. Then we will see how this algorithm can be used to overcome the difficulty of anonymity.

The LinkName algorithm uses the link-register model, with read/write atomicity. Each node u has one register per neighbour v: $name_{uv}$, that is the name of the link (u, v) given by u. This name can also be seen as the name of node v given by u.



The algorithm ensures that eventually every node u will give a unique name to each of its neighbours. More formally, we require the following specification $\mathcal{LN}$ for LinkName:

SPECIFICATION 3. *(Specification $\mathcal{LN}$)*
For a graph $G = (V, E)$, $\mathcal{LN}$ is:

- $\forall u \in V, \forall i \in \{1, \ldots, |N(u)|\}, \exists v \in N(u) : \ name_{uv} = i$

Algorithm 4 LinkName

$\triangleright(R_0)$
if $\neg(\forall i \in \{1, \ldots, |N(u)|\}, \exists v \in N(u) : \ name_{uv} = i)$ **then**
    Rename all $name_{uv}$ *from 1 to* $|N(u)|$

Algorithm LinkName [Doloo] satisfies the following theorem:

THEOREM 8. *Under the adversarial distributed daemon and with the read/write atomicity, the LinkName algorithm is self-stabilizing and silent for the specification $\mathcal{LN}$, and it reaches a stable configuration in $O(m)$ moves.*

Proof of 8. We start by giving the complexity of algorithm LinkName in term of moves. Node u executes rule R_0 at most once. During this execution, u makes at most 2 moves per edge (u, v): one reading-move to check whether $name_{uv} = i$ and at most one writing-move to update $name_{uv}$. Thus the maximum number of moves in any execution of the algorithm is $4m$ moves.

In a stable configuration, the guard of rule R_0 is false for every node. Thus, in every stable configuration of LinkName, the specification $\mathcal{LN}$ holds.

Since, under the adversarial distributed daemon and with the read/write atomicity, any execution of the link-name algorithm LinkName reaches a stable configuration in at most $4m$ moves, we obtain Theorem 8. 8

The link-register model allows to locally distinguish the links incident to a node. However, this model does not build distance-2 unique identifiers. Thus, the impossibility result proved by Manne et al. [19] still holds under the link-register model and then we still need a probabilistic algorithm to solve the maximal matching problem under the link-register model. The solution proposed in this paper is to compose the LinkName algorithm with a rewritten version of AnonyMatch algorithm that can use the LinkName registers.

We now give a systematic way to rewrite an anonymous matching algorithm using registers of LinkName in order to avoid instructions that violate the anonymous assumption. For example, if we consider the AnonyMatch algorithm, we

would like to see the Marriage rule

$$\textbf{if } (\beta_u = \perp) \wedge (\exists v \in N(u) : \beta_v = u) \textbf{ then } \beta_u := v$$

rewritten as:

$$\textbf{if } (\beta_u = \perp) \wedge (\exists v \in N(u) : \beta_v = name_{vu}) \textbf{ then } \beta_u := name_{uv}$$

When character u or v appears in the algorithm, it can represent either node u or the identifier of node u. Only the first case is meaningful in an anonymous system. Other occurrences of u (resp. v) should be replaced by $name_{uv}$ (resp. $name_{vu}$). Syntactically, the first case corresponds to the use of u and v as indices in a broad meaning – *i.e.*, as indices of a local variable (as in β_u) or to designate a node's neighbourhood (as in $N(u)$) –, or when quantified (as in $\forall v \in N(u)$), while the second case is when u and v are used as identifiers, to be compared with other identifiers. Thus, u and v should be maintained when used as indices (or arguments of N) and when quantified; they should be replaced with $name_{uv}$ and $name_{vu}$ respectively when they are compared (with comparison operators $=, <, \ldots$)

We give above the algorithm $\mathcal{A}$NONY$\mathcal{M}$ATCH fully rewritten using this rule:

Algorithm 5 $\mathcal{A}$NONY$\mathcal{M}$ATCH2 - Maximal matching algorithm in anonymous graphs

▷**Marriage rule**
if $(\beta_u = \perp) \wedge (\exists v \in N(u) : \beta_v = name_{vu})$ **then**
  $\beta_u \leftarrow name_{uv}$

▷**Seduction rule**
if $(\beta_u = \perp) \wedge (\forall v \in N(u) : \beta_v \neq name_{vu}) \wedge (\exists x \in N(u) : \beta_x = \perp)$ **then**
  if $choose(\{0, 1\}) = 1$ **then**
    $\beta_u \leftarrow choose(\{x \in N(u) : \beta_x = \perp\})$
  else
    $\beta_u \leftarrow \perp$

▷**Abandonment rule**
if $(\exists v \in N(u) : \beta_u = name_{uv} \wedge \beta_v \neq name_{vu} \wedge \beta_v \neq \perp)$ **then**
  $\beta_u \leftarrow \perp$

Having defined algorithms $\mathcal{A}$NONY$\mathcal{M}$ATCH2 and $\mathcal{L}$INK$\mathcal{N}$AME, we would like to compose them to give a unified self-stabilizing algorithm. However, this cannot be done in a straightforward way. Indeed, the two algorithms use different communication and atomicity models: algorithm $\mathcal{A}$NONY$\mathcal{M}$ATCH2 assumes the state model with the guarded rule atomicity, while $\mathcal{L}$INK$\mathcal{N}$AME assumes the link-register model with the read/write atomicity. For this composition, we keep both models. So $\mathcal{A}$NONY$\mathcal{M}$ATCH2 and $\mathcal{L}$INK$\mathcal{N}$AME run in the same execution, under these two different models.

We cannot directly apply the composition result of Dolev et al. [DIM89] since authors assume the same model for their composition. However, we can use similar arguments:

1) LinkName neither reads nor writes in variables of AnonyMatch2 while AnonyMatch2 only reads in registers of LinkName.
2) LinkName stabilizes independently of AnonyMatch2.

Concerning AnonyMatch2, it only has been proved under the state model (AnonyMatch proof in previous section) while it uses *registers* from LinkName. However we can notice that LinkName is silent thus the value of these registers will eventually not change. Moreover, these registers are used to give name to nodes and then when LinkName is stabilized, they actually give correct names to nodes and so AnonyMatch2 can behave correctly as if it was AnonyMatch. So the proof that has been done for AnonyMatch is still valid for AnonyMatch2.

Thus once LinkName is stabilized and reaches a stable configuration, AnonyMatch2 eventually stabilizes under the state model and the guarded rule atomicity.

Note that these arguments can be applied to any anonymous matching that would use our rewritten rule. Then it makes sense to assume in an anonymous matching that a node can know if one of its neighbour points to it or to some other node.

4.6 CONCLUSION

A distributed daemon, in an anonymous network, can exploit symmetries and manage to reproduce them from round to round, making it impossible to solve many basic problems in these conditions. Therefore, writing distributed algorithms for anonymous networks often requires extra techniques that can be avoided in identified networks. Now, anonymity is often a requirement in practical systems, either because nodes have too few capabilities, or for safety reasons.

The work presented in this chapter is a randomized distributed algorithm that computes a maximal matching in an anonymous networks under an adversarial distributed daemon. This problem cannot be solved without randomisation, as shown by [MMPT09]: indeed, the distributed daemon can, at each step, pick up several nodes that will have to abandon their wedding projects.

Thus, solving this problem necessitates that the node can take some decisions independently from the daemon. The general randomisation scheme of Goddard et al. [9], when applied to an algorithm working under a sequential daemon, leads to a much worse expected complexity of $O(mn^3)$. Indeed, the general requirement met by this scheme imposes to break "long-range" symmetries while this problem requires only distance 2 symmetry breaking.

The key issue is thus to allow the nodes to break symmetries: the proposed algorithms illustrates that, when identifiers cannot be used as a symmetry-breaking tool,

randomisation can be an alternative. In algorithm $\mathcal{A}$NONY$\mathcal{M}$ATCH, randomisation allows nodes to probabilistically refuse activation, thus breaking the symmetry that the daemon tries to impose. Thus, we prove, with this algorithm, that randomisation allows to solve this problem, and that the cost of anonymity is a constant factor, as we do not change the order of growth of the complexity (from $O(m)$ to $O(n^2)$).

The proposed randomisation uses one bit of randomness per Single node activation (and thus less than one bit of randomness per node activation; the random choice of the neighbour to seduce does not intervene in any proof and can be replaced by an arbitrary choice of the neighbour). Moreover, the proof of Theorem 1 establishes that no more than an expected $4n$ single nodes can be activated in an execution, so that an execution requires $4n$ random bits on average, and any node requires a constant expected 4 random bits to achieve Maximal Matching. This quantity is to be compared to the $\log n$ bits necessary to achieve a proper naming (this discrepancy can be easily explained by the fact that a distance-2 colouring would be sufficient, rather than a complete naming).

Last, we show that the classical state model implies the assumption that anonymous nodes can know whether their neighbours point to them, and that this hypothesis is not contradictory. Indeed, we propose a simple rewriting scheme that allows to transform a state model algorithm in a link-register one, in which anonymous nodes eventually know the name each neighbour attributes to the link joining them. This transformation only entails a constant factor on the complexity.

5 A polynomial 2/3– approximation of the maximum matching problem

In this chapter we move on to the maximum version of the matching problem in identified networks. We devise a self-stabilizing polynomial algorithm for a 2/3– approximation of the problem under the adversarial distributed daemon. We also prove, by exhibiting a sub-exponential execution in term of moves of the ExpoMatch algorithm by Manne *et al.* [MMPT11], that our new algorithm improves on the latest result in the literature. This work is published in [CMMP16].

5.1 INTRODUCTION

In sequential algorithms, the maximum matching problem and its weighted generalization are known to be polynomial (see [Edm87] [Gal86]). Some (almost) linear time approximation algorithms for the maximum weighted matching problem have been well studied [DH03, Pre99], nevertheless these algorithms are not distributed. They are based on a simple greedy strategy using *augmenting paths*. An *augmenting path* is a path, starting and ending in an unmatched node, and where every other edge is either unmatched or matched; *i.e.* for each consecutive pair of edges, exactly one of them must belong to the matching. It is well known [HK73] that given a graph $G = (V, E)$ and a matching $M \subseteq E$, if there is no augmenting path of length $2k - 1$ or less, then M is a $\frac{k}{k+1}$–approximation of the maximum matching. See [DH03] for the weighted version of this theorem. The greedy strategy in [DH03, Pre99] consists in finding all augmenting paths of length ℓ or less and by switching matched and unmatched edges of these paths in order to improve the maximum matching approximation.

In this chapter, we present a self-stabilizing algorithm for finding a 1-maximal matching that uses the greedy strategy presented above. Our algorithm stabilizes after $O(m \times n^2)$ moves under the adversarial distributed daemon.

For the maximum matching problem, self-stabilizing algorithms have been designed for particular topologies. In anonymous tree networks, a self-stabilizing algorithms converging in $O(n^4)$ moves under the sequential adversarial daemon

is given by Karaata and Saleh [KS00]. Recently, Datta *et al.* [DLM16] improve this result, and give a silent self-stabilizing protocol that converges in $O(n^2)$ moves. For anonymous bipartite networks, a self-stabilizing algorithm converging in $O(n^2)$ rounds under the sequential daemon is given by Chattopadhyay et al. [CHS02]. So, this algorithm converges in $\Omega(n^3)$ moves (since one round corresponds to at least n moves). Let us restate that in unweighted or weighted general graphs, self-stabilizing algorithms for computing maximal matching have been designed in various models (anonymous network [AI15] or not [TH11b], see [GK10] for a survey). For an unweighted graph, Hsu and Huang [HH92] gave the first self-stabilizing algorithm and proved a bound of $O(n^3)$ on the number of moves under a sequential adversarial daemon. Hedetniemi *et al.* [HJS01] completed the complexity analysis proving a $O(m)$ move complexity. Manne *et al.* [MMPT09] gave a self-stabilizing algorithm that converges in $O(m)$ moves under a distributed adversarial daemon. The results presented in the previous chapter Cohen *et al.* [CLM+16] extend this result and propose a randomized self-stabilizing algorithm for computing a maximal matching in an anonymous network. The complexity is $O(n^2)$ moves with high probability, under the adversarial distributed daemon.

Manne *et al.* [MMPT11] and Asada and Inoue [AI15] presented some self-stabilizing algorithms for finding a 1-maximal matching. Manne *et al.* gave an exponential upper bound on the stabilization time of their algorithm ($O(2^n)$ moves under a distributed adversarial daemon). However, they didn't show that this upper bound is tight. Here, we prove that this lower bound is sub-exponential by exhibiting an execution of $\Omega(2^{\sqrt{n}})$ moves before stabilization. Asada and Inoue [AI15] gave a polynomial algorithm but under the adversarial sequential daemon. Recently, Inoue *et al.* [IOT16] gave a modified version of [AI15] that stabilizes after $O(m)$ moves under the distributed adversarial daemon for networks without cycle whose length is a multiple of three.

In a weighted graph, Manne and Mjelde [MM07] presented the first self-stabilizing algorithm for computing a weighted matching of a graph with a $\frac{1}{2}$-approximation of the optimal solution. They established that their algorithm stabilizes after at most an exponential number of moves under any adversarial daemon (*i.e.*, sequential or distributed). Turau and Hauck [TH11b] gave a modified version of the previous algorithm that stabilizes after $O(nm)$ moves under any adversarial daemon. Figure 5.1 compares features of the aforementioned algorithms and our result.

We are then interested in the following problem: how to efficiently build a 1-maximal matching in an identified graph with a general topology, using an adversarial distributed daemon and in a self-stabilizing way? Here, we present two algorithms solving this problem. The first one is the well-known algorithm from Manne *et al.* [MMPT11] that was the only one until now that solved this problem. The second algorithm is our contribution. We show that the Manne *et al.* algorithm reaches a sub-exponential complexity while we prove that our algorithm is polynomial (in $O(m \times n^2)$).

Matching	Topology	Identifiers	Daemon	Complexity (moves)	Work
Maximum	Tree Bipartite	Global Anonymous	Adver. Sequential	$O(n^2)$ $\Omega(n^3)$	[KS00, DLM16] [CHS02]
Maximal	Arbitrary	Global	Adver. Sequential Adver. Distributed	$O(m)$ $O(m)$	[HH92, HJS01] [MMPT09]
		Anonymous	Adver. Sequential Adver. Distributed	$O(n^2)$ $O(n^2)$ whp	[HH92] [CLM+16]
1-Maximal	Arbitrary without cycle with multiple of 3 length	Anonymous	Adver. Sequential Adver. Distributed	$O(m)$ $O(m)$	[AI15] [IOT16]
	Arbitrary	Global	Adver. Distributed	$\mathbf{\Omega(2^{\sqrt{n}})}$ $\mathbf{O(m.n^2)}$	**Here** **Here**

FIGURE 5.1
Best results in maximum matching approximation. In bold, our contributions.

5.2 COMMON STRATEGY TO BUILD A 1-MAXIMAL MATCHING

In this chapter, we present two algorithms. The first one, denoted by ExpoMatch, is the Manne *et al.* algorithm [MMPT11]. The second one, called PolyMatch, is the main contribution. These two algorithms share several elements and this section is devoted to give these main common points.

Both algorithms operate on an undirected graph, where every node has a unique identifier. They also assume an adversarial distributed daemon and that there exists an already built maximal matching, noted $\mathcal{M}$. Based on $\mathcal{M}$, the two algorithms build a 1-maximal matching. To perform that, nodes search and delete any 3-augmenting paths they find in $\mathcal{M}$.

5.2.1 *3-augmenting path*

An augmenting path is a path in the graph, starting and ending in an unmatched node, and where every other edge is either unmatched or matched.

DEFINITION 38. *Let $G = (V, E)$ be a graph and M be a maximal matching of G. (x, u, v, y) is a* 3-augmenting path *on (G, M) if:*

a) *(x, u, v, y) is a path in G (so all nodes are distinct);*
b) *$\{(x, u), (v, y)\} \subset E \setminus M$;*
c) *$(u, v) \in M$*

Let us consider the example in Figure 5.2.(a). In this figure, u and v are matched nodes and x, y are unmatched nodes. The path (x, u, v, y) is a 3-augmenting path.

Once an augmenting path is detected, nodes rearrange the matching accordingly, *i.e.*, transform this path with one matched edge into a path with two matched edges (see Figure 5.2.(b)). This transformation leads to the deletion of the augmenting path and increases by one the cardinality of the matching. Both algorithm will stabilize when there are no augmenting paths of length three left. Thus the hypothesis of Karps's theorem [HK73] eventually holds, giving a $\frac{2}{3}$–approximation of the maximum matching (and so a 1-maximal matching).

(a) A 3-augmenting path (one matched edge)

(b) The path after being expoited. (two matched edges)

FIGURE 5.2 How to exploit a 3-augmenting path?

5.2.2 *The underlying maximal matching*

In the rest of this chapter, $\mathcal{M}$ is the underlying maximal matching. This underlying matching is locally expressed by variables m_v for each node v. If $(u, v) \in \mathcal{M}$ then u and v are *matched nodes* and we have: $m_u = v \wedge m_v = u$. If u is not incident to any edge in $\mathcal{M}$, then u is a *single node* and $m_u = null$. For a set of nodes A, we define $single(A)$ and $matched(A)$ as the sets of single and matched nodes in A, accordingly to the underlying maximal matching $\mathcal{M}$. Since we assume $\mathcal{M}$ to be stable, a node membership in $matched(V)$ or $single(V)$ will not change throughout an execution, and each node u can use the value of m_u to determine which set it belongs to.

Note that $\mathcal{M}$ can be built with any silent self-stabilizing maximal matching algorithm that works for general graphs and with an adversarial distributed daemon. We can then use, for instance, the self-stabilizing maximal matching algorithm from [MMPT09] that stabilizes in $O(m)$ moves. Observe that this algorithm is silent, meaning that the maximal matching remains constant once the algorithm has stabilized.

5.2.3 *Augmenting paths detection and exploitation*

Both algorithms ExpoMatch and PolyMatch are based on two phases for each edge (u, v) in $\mathcal{M}$: (1) *detecting* augmenting paths and (2) *exploiting* the detected augmenting paths. Node u keeps track of four variables. The pointer p_u is used to define the final matching. The variables α_v, β_u are used to detect augmenting paths and contain single neighbours of u. Also, s_u is a boolean variable used for the augmenting path exploitation. We will see in section 5.5 that algorithm PolyMatch uses a fifth variable named end_u.

In the rest of the paper, we will call $\mathcal{M}^+$ the final 1-maximal matching built by any of the two algorithms. $\mathcal{M}^+$ is defined as follows:

DEFINITION 39. *The built set of edges is:*

$$\mathcal{M}^+ = \{(u, v) \in \mathcal{M} : p_u = p_v = null\} \cup \{(a, b) \in E \setminus \mathcal{M} : p_a = b \wedge p_b = a\}$$

The first set in the union is pairs of nodes that do not perform any rematch. These pairs come from $\mathcal{M}$. The second set in the union is pairs of nodes that were not matched together in $\mathcal{M}$, but after a 3-augmenting path detection and exploitation, they matched together.

AUGMENTING PATH DETECTION First, every pair of matched nodes u, v (v=m_u and u=m_v) tries to find single neighbours they can rematch with. These single neighbours have to be *available*, in particular, they should not be married in a final way with another matched node. We will see in the next sections, that the meaning of being available is not the same in PolyMatch and ExpoMatch. We say that a single node x is a *candidate* for a matched node u if x is an available single neighbour of u. Note that u and v need to have a sufficient number of candidates to detect a 3-augmenting path: each node should have at least one candidate and the sum of the number of candidates for u and v should be at least 2. In both algorithms, the *BestRematch* predicate is used to compute candidates of a matched node u, writing in α_u and β_u. Then, the condition below is used in both algorithms – in the *AskFirst* predicate – to ensure the number of candidates is sufficiently high to detect if u belongs to a 3-augmenting path.

$$(\alpha_u \neq null \wedge \alpha_{m_u} \neq null) \wedge (2 \leqslant Unique(\{\alpha_u, \beta_u, \alpha_{m_u}, \beta_{m_u}\}) \leqslant 4)$$

where $Unique(A)$ returns the number of unique elements in the multi-set A.

AUGMENTING PATH EXPLOITATION The exploitation is done in a sequential way. First, two nodes matched together u and v agree on which one starts to build a rematch and which one ends. This local consensus is done using *AskFirst* and *AskSecond* predicates. Observe that these predicates are exactly the same in both algorithms. These predicates use the local state of u and v to assign a role to these two nodes. If *AskFirst(u)* is *True* then u starts to rematch and v ends. If *AskSecond(u)* is *True* then v starts to rematch and u ends.

Observe that there are only three distinct possible values for the quadruplet ($AskFirst(u)$, $AskSecond(u)$, $AskFirst(v)$, $AskSecond(v)$) for any couple $(u, v) \in \mathcal{M}$ and whatever the α and β values are. These are: ($null$, $null$, $null$, $null$) or (x, $null$, $null$, y) or ($null$, x, y, $null$), with x and y are two distinct single nodes. The first case means that there is no 3-augmenting path that contains the couple (u, v). The two other cases mean that (x, u, v, y) is a 3-augmenting path. The second case occurs when $x < y$, otherwise we are in the third case. Node u is said to be *First* if $AskFirst(u) \neq null$. In the same way, u is *Second* if $AskSecond(u) \neq null$. So, if a 3-augmenting path is detected though (u, v), the roles of u and v depend on the identifiers of single nodes (candidates) in the augmenting path, *i.e.*, u is *First* iff its single neighbour in the augmenting path has a smaller identifier that the single neighbour of v in the augmenting path.

5.2.4 *Graphical convention*

We will follow the above conventions in all the figures: matched nodes are represented with thick circles and single nodes with thin circles. Node identifiers are indicated inside the circles. Moreover, all edges that belong to the maximal matching $\mathcal{M}$ are represented with a thick line, whereas the other edges are

represented with a simple line. In the same way, all matched nodes are represented with a thick line, whereas single nodes are represented with a simple line. We illustrate the use of the p-values by an arrow, and the absence of the arrow or symbol 'T' mean that the p-value of the node equals to null. A prohibited value is first drawn in grey, then scratched out in black. For instance, in Figure 5.3, node 10 is single, nodes 9 and 8 are matched and the edge $(8, 9) \in \mathcal{M}$.

5.3 DESCRIPTION OF THE ALGORITHM EXPOMATCH

In this section, we precisely describe the algorithm ExpoMatch [MMPT11]. The algorithm itself is shown in Algorithm 7.

5.3.1 *Augmenting paths detection and exploitation*

AUGMENTING PATH DETECTION In this algorithm, a single node x is a *candidate* for a matched node u if it is not involved in another augmenting path exploitation, *i.e.*, if $p_x = null \vee p_x = u$.

AUGMENTING PATH EXPLOITATION A 3-augmenting path is exploited in two phases. These two phases are performed in a sequential way. Recall that node u is said to be *First* if $AskFirst(u) \neq null$ and node u is *Second* if $AskSecond(u) \neq null$. Let us consider two nodes u and v such that $(u, v) \in \mathcal{M}$. Let us assume that u and v detects an augmenting path.

1) The *First* node starts : Exactly one node among u and v attempts to rematch with one of its candidates. This phase is complete when the first node, let say u, is such that $s_u = True$ and this indicates to the *Second* node (v) that the first phase is over.
2) The *Second* node continues: only when the first node succeeds will the second node attempt to rematch with one of its candidates.
   a) If this also succeeds, the exploitation is done and the augmenting path is said to be *fully exploited*
   b) Otherwise the rematch built by the *First* node is deleted and candidates α and β are computed again, allowing then the detection of some new augmenting paths.

5.3.2 *Rules description*

There are four rules for matched nodes. The *Update* rule is the rule with the highest priority. This rule allows a matched node to update its α and β variables, using the *BestRematch* predicate. Then, predicates *AskFirst* and *AskSecond* are used to define the role the node will have in the 3-augmenting path exploitation.

If the node is *First* (resp. *Second*), then it will execute *MatchFirst* (resp. *MatchSecond*) several times for this 3-augmenting path exploitation. The *ResetMatch* rule is performed to reset bad initialization and also to reset an augmenting path exploitation that did not terminate. For instance, this case happens when the single candidate of the *Second* node rematch with some other node in the middle of the exploitation path process.

Let us consider $(u, v) \in \mathcal{M}$ and assume that u and v detect an augmenting path with u is *First*. The *MatchFirst* rule is used by u to build its rematch. The rule is performed a first time by u to propose a rematch to its candidate x (u sets p_u to x). Then, if x accepts ($p_x = u$), u performs this rule a second time to communicate to v that its rematch attempt is a success (u sets s_u to $True$). The *MatchSecond* rule is used by the node v to build its rematch. This rule can only be performed if $s_u = True$. Then, the rule is performed once by v to propose a rematch to its candidate y (v sets p_v to y). Then, if y accepts ($p_y = v$), the path is fully exploited and will not change during the rest of the execution.

There is only one rule for single nodes, called *SingleNode*. Recall that all neighbours of a single node are matched, since $\mathcal{M}$ is a maximal matching. A single node should always point to its smallest neighbour that points to it. This rule allows to point to such a neighbour but also to reset a bad p-value to *null*. Observe that a single node x cannot perform this rule if $p_{p_x} = x$, which means that if x point to some neighbour that points back to x, then x is locked.

Algorithm 6 Functions used in the ExpoMatch algorithm

function BestRematch(v)
  $a := Lowest\ \{v \in single(N(u)) \wedge (p_v = null \vee p_v = u)\}$
  $b := Lowest\ \{v \in single(N(u)) \setminus \{a\} \wedge (p_v = null \vee p_v = u)\}$
  return (a, b)

function AskFirst(u)
  if $\alpha_u \neq null \wedge \alpha_{m_u} \neq null \wedge 2 \leqslant Unique(\{\alpha_u, \beta_u, \alpha_{m_u}, \beta_{m_u}\}) \leqslant 4$ **then**
    if $\alpha_u < \alpha_{m_u} \vee (\alpha_u = \alpha_{m_u} \wedge \beta_u = null) \vee (\alpha_u = \alpha_{m_u} \wedge \beta_{m_u} \neq null \wedge u < m_u)$ **then**
      return α_v
    return *null*

function AskSecond(u)
  if $AskFirst(m_u) \neq null$ **then**
    return $Lowest(\{\alpha_u, \beta_u\} \setminus \{\alpha_{m_u}\})$
  return *null*

$Unique(A)$ returns the number of unique elements in the multi-set A.
$Lowest(A)$ returns the node in A with the lowest identifier. If $A = \varnothing$, then $Lowest(A)$ returns *null*.

Algorithm 7 ExpoMatch algorithm

▷**SingleNode** ▷ Rule for each node u in **single**(**V**)
if $(p_u = null \wedge Lowest\{v \in N(u) \mid p_v = u\} \neq null) \vee p_u \notin matched(N(u)) \cup \{null\} \vee (p_u \neq null \wedge p_{p_u} \neq u)$ **then**
$p_u := Lowest\{v \in N(u) \mid p_v = u\}$

▷**Update** ▷ Rules for each node u in **matched**(**V**)
if $(\alpha_u > \beta_u) \vee (\alpha_u, \beta_u \notin single(N(u)) \cup \{null\}) \vee (\alpha_u = \beta_u \wedge \alpha_u \neq null) \vee p_u \notin single(N(u)) \cup \{null\} \vee ((\alpha_u, \beta_u) \neq \text{BestRematch}(u) \wedge (p_u = null \vee p_{p_u} \notin \{u, null\}))$ **then**
$(\alpha_u, \beta_u) := \text{BestRematch}(u)$
$(p_u, s_u) := (null, false)$

▷**MatchFirst**
Let $x = \text{AskFirst}(u)$
if $x \neq null \wedge (p_u \neq x \vee s_u \neq (p_{p_u} = u))$ **then**
$p_u := x$
$s_v := (p_{p_u} = u)$

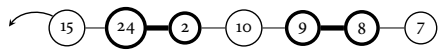
▷**MatchSecond**
Let $y = \text{AskSecond}(u)$
if $y \neq null \wedge s_{m_u} = true \wedge p_u \neq y$ **then**
$p_u := y$

▷**ResetMatch**
if $\text{AskFirst}(u) = \text{AskSecond}(u) = null \wedge (p_u, s_u) \neq (null, false)$ **then**
$(p_u, s_u) := (null, false)$

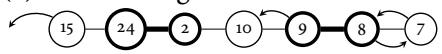
5.3.3 *An execution example of the ExpoMatch algorithm*

Now, we give a possible execution of Algorithm ExpoMatch under a distributed adversarial daemon. Figure 5.3.(a) shows the initial configuration of the execution. The topology is a path of seven vertices. The underlying maximal matching represented by bold edges contains two edges $(24, 2)$ and $(9, 8)$. Then nodes 24, 2, 9 and 8 are *matched* nodes (in $matched(V)$) and nodes 15, 10 and 7 are *single* nodes (in $single(V)$). There are two 3-augmenting paths: $(15, 24, 2, 10)$ and $(10, 9, 8, 7)$.

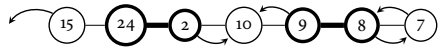
THE INITIAL CONFIGURATION (FIGURE 5.3.(A)) In the initial configuration, we assume that α-*values* and β-*values* are defined as follows: $(\alpha_8, \beta_8) = (7, null)$, $(\alpha_9, \beta_9) = (10, null)$ and $(\alpha_{24}, \beta_{24}) = (15, null)$ and $(\alpha_2, \beta_2) = (10, null)$. We also assume all *s-values* are well defined: $s_8 = s_9 = s_2 = s_{24} = false$. At this step, node 15



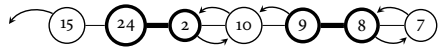
(a) Initial configuration.



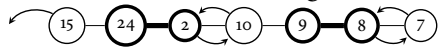
(b) The exploitation of $(10, 9, 8, 7)$ starts.



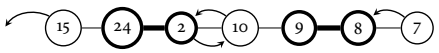
(c) Node 2 executes a *MatchFirst* move.



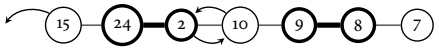
(d) Node 10 executes a *SingleNode* move.



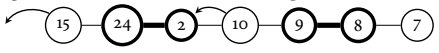
(e) Node 9 executes an *Update* move.



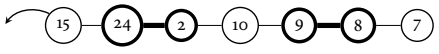
(f) Node 8 executes a *ResetMatch* move.



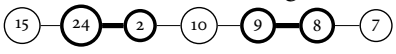
(g) Node 7 executes a *SingleNode* move.



(h) Node 2 executes a *ResetMatch* move.



(i) Node 10 execute a *SingleNode* move.



(j) Node 15 executes a *SingleNode* move.

FIGURE 5.3
An execution of Algorithm ExpoMatch

has its *p-values* such that: $p_{15} \notin \{null, 24\}$.

Observe that in the initial configuration, we only have two wrong values: p_{15} and α_{24}. We are going to show that these two faulty nodes can generate the destruction of an augmenting path exploitation, even if this exploited path do not contain any faulty node. This scenario is the fundamental reason why ExpoMatch algorithm is sub-exponential.

THE 3-AUGMENTING PATH EXPLOITATION OF $(10, 9, 8, 7)$ STARTS (FIGURE 5.3.(A-B)) Nodes 8 and 9 can start to exploit their augmenting path: node 8 is *First* because $\alpha_8 < \alpha_9$, so node 8 executes a *MatchFirst* move and sets $p_8 = 7$. At this point, node 8 waits for an answer of node 7. Node 7 accepts to take part of this path exploitation setting $p_7 = 8$ (by performing a *SingleNode* rule). Afterwards, node 8 can tell node 9 that it can start its exploitation too. Thus node 8 executes again a *MatchFirst* move and sets $s_8 = True$. Now, node 9 can start his exploitation. Assume that it does by executing a *MatchSecond* move. Then node 9 waits for an answer of node 10 and we are in configuration drawn in Figure 5.3.(b).

THE 3-AUGMENTING PATH EXPLOITATION OF $(15, 24, 2, 10)$ STARTS (FIGURE 5.3.(B-D)) Now, we focus on the other 3-augmenting path $(15, 24, 2, 10)$. At this moment, since $2 \leqslant Unique(\{\alpha_{24}, \beta_{24}, \alpha_2, \beta_2\}) \leqslant 4$, node 2 detects a 3-augmenting path and starts to exploit it. Since node 2 is *First* ($AskFirst(2) = 10$), node 2 can execute a *MatchFirst* move. Let us assume it does (see Figure 5.3.(c)).

Since both nodes 9 and 2 are pointing to node 10, node 10 can choose the node to match with from these two nodes. Node 10 makes this choice by executing an *SingleNode* move: since $Lowest\{u \in N(10) \mid p_u = 10\} = 2$, node 10 chooses node 2 (see Figure 5.3.(d)).

THE 3-AUGMENTING PATH $(10, 9, 8, 7)$ EXPLOITATION IS DESTROYED (FIGURES 5.3.(D-G)) Node 9 considers that node 10 belongs to another 3-augmenting

path because $p_{10} \notin \{null, 9\}$. Moreover, since $(\alpha_9, \beta_9) \neq BestRematch(9)$, node 9 can execute an *Update* move. Let us assume it does. Figure 5.3.(e) shows the configuration obtained after this move: $(\alpha_9, \beta_9) = (null, null)$ and $(p_9, s_9) = (null, false)$. This will cause $AskFirst(8) = AskSecond(8) = null$. Then node 8 executes a *ResetMatch* move (see configuration after this move in Figure 5.3.(f)). This will cause node 7 to execute a *SingleNode* move and sets $p_7 = null$ as seen in Figure 5.3.(g).

FOCUS ON THE 3-AUGMENTING PATH $(15, 24, 2, 10)$ (FIGURES 5.3.(G-J))
Let us assume now that the faulty node 24 is activated. It executes an *Update* move (because $(\alpha_{24}, \beta_{24}) \neq BestRematch(24)$) and sets $(\alpha_{24}, \beta_{24}) = (null, null)$. After this move, node 2 detects that it does not belong to any 3-augmenting path since $AskFirst(2) = AskSecond(2) = null$. So, node 2 executes a *ResetMatch* move and sets $(p_2, s_2) = (null, false)$ (see Figure 5.3.(h)). Afterwards, node 10 executes a *SingleNode* move to set p_{10} to *null* (see Figure 5.3.(i)). Now, only node 15 is activable and it executes a *SingleNode* move in order to set p_{15} to *null* (see Figure 5.3.(j)). At this moment, the two exploitation processes for the two 3-augmenting paths can start again.

5.4 THE EXPOMATCH ALGORITHM IS SUB-EXPONENTIAL.

In this section, we exhibit an execution of length 2^N in a *chosen* graph having $\Theta(N^2)$ nodes. To do that, we define, under some conditions, how to translate a configuration into a binary integer. Then, we give an execution where all configurations corresponding to integers from 0 to $2^N - 1$ appear. This gives us an execution of length $\Omega(2^N)$.

5.4.1 *State of a matched edge*

A bit in the binary representation of a given configuration corresponds to a particular state of the nodes in a 3-augmenting path. More precisely, according to the p-values of these nodes, the associated bit of the path will be 0, 1 or *undef*. Figure 5.6 represents an instance of the chosen graph for $N = 4$. Observe that any matched node only has one single neighbor. This property will hold for any N. Thus, a 3-augmenting path can be determined by its matched edge.

DEFINITION 40. *(State of a matched edge)*
Let $e = (u, v)$ be an edge in the maximal matching $\mathcal{M}$ such that u (resp. v) has one single neighbor x (resp. y). Assume $y < x$. Edge e is said to be:

- in state OFF *if $p_x = null$, $p_u = null$, $p_v = null$ and $p_y = null$.*
- in state ALMOSTOFF *if $p_x \notin \{null, u\}$, $p_u = null$, $p_v = null$, and $p_y = null$.*
- in state ON *if $p_x = null$, $p_u = x$, $p_v = y$ and $p_y = v$.*

- in state AlmostOn *if* $p_x \notin \{null, u\}$, $p_u = x$, $p_v = y$ *and* $p_y = v$.

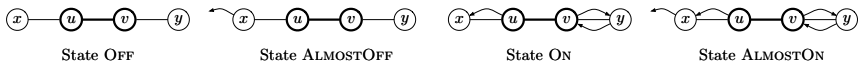


FIGURE 5.4
Four states of an edge

Note that a matched edge can be in none of the states presented below.

An illustration of Definition 40 can be seen in Figure 5.4. Moreover, if we consider the edge (8, 9) in Figure 5.3, we have: the edge is in state Off in Figure 5.3.(a), in state On in Figure 5.3.(b), in state AlmostOn in Figure 5.3.(d) and in state AlmostOff in Figure 5.3.(g). Finally, there is no state associated to the edge in Figure 5.3.(f)

The states of an edge represent the different steps of an augmenting path exploitation. Now, we exhibit an execution to switch an edge (u, v) from state Off to state On in Lemma 5 and then, from state AlmostOn to state AlmostOff in Lemma 6.

LEMMA 5. *Let $e = (u, v)$ be an edge in the maximal matching $\mathcal{M}$ such that u (resp. v) has one single neighbor x (resp. y). Assume $y < x$. Let C be a configuration where:*

$$e \text{ is in state Off and } v = min(\{w \in N(y) : p_w = y\} \cup \{v\}).$$

There exists a finite execution starting in C and ending in D such that:

- *(i) only nodes u, v and y make moves between C and D and*
- *(ii) edge e is in state On in D.*

Proof of 5. We describe a finite execution starting in C and ending in D that allows to switch edge (u, v) from state Off to state On and where only nodes u, v and y make moves. Nodes u and v belong to a 3-augmenting path in C since $p_x = p_y = null$ by assumption. If $\alpha_u \neq x$, then node u executes an *Update* move and sets $(\alpha_u, \beta_u) = (x, null)$. If $\alpha_v \neq y$, then node v executes an *Update* move and sets $(\alpha_v, \beta_v) = (y, null)$.

Now, the variables α_u and α_v are well defined. Since $y < x$, we have $AskFirst(v) = y$ and $AskSecond(u) = x$. So node v executes a *MatchFirst* move and sets $p_v = y$. Let $C_1 \mapsto C_2$ be the transition where v makes this *MatchFirst* move. Observe that only u and v made some moves from C to C_2. Moreover, $u \notin N(y)$ since u has only one single neighbor that is x. Thus $v = min(\{w \in N(y) : p_w = y\} \cup \{v\})$ still holds in C_2 and so, node y chooses node v to match with by executing a *SingleNode* move. Finally, node u is eligible to execute a *MatchSecond* move and it then points to node x. The edge (u, v) is now in state On. 5

Now, we exhibit an execution to switch edge (u, v) from state AlmostOn to state AlmostOff.

LEMMA 6. *Let $e = (u, v)$ be an edge in the maximal matching $\mathcal{M}$ such that u (resp. v) has one single neighbor x (resp. y). Assume $y < x$. Let C be a configuration where:*

$$e \text{ is in state AlmostOn and } \{w \in N(y) : p_w = y\} = \{v\}$$

There exists a finite execution starting in C and ending in D such that:

- *(i) only nodes u, v and y make moves between C and D and*
- *(ii) edge e is in state* AlmostOff *in D.*

Proof of 6. We describe a finite execution starting in C and ending in D that allows to switch edge (u, v) from state AlmostOn to state AlmostOff and where only nodes u, v and y make moves. Since edge (u, v) is in state AlmostOn, then $p_x \notin \{null, u\}$ and so $BestRematch(u) = (null, null)$. If $(\alpha_u, \beta_u) \neq (null, null)$ then node u executes an *Update* move. Otherwise, $AskFirst(u) = AskSecond(u) = null$ and, since $p_u \neq null$, u executes a *ResetMatch* move. In both cases, after the move, $(p_u, s_u) = (null, false)$ and $(\alpha_u, \beta_u) = (null, null)$.

$\alpha_u = null$ implies $AskFirst(v) = null$, and $AskFirst(u) = null$ implies $AskSecond(v) = null$. Moreover, since $p_v \neq null$, v executes a *ResetMatch* move and sets $p_v = null$. Let $C_1 \mapsto C_2$ be the transition where v makes this *ResetMatch* move. Since $\{w \in N(y) : p_w = y\} = \{v\}$ holds in the configuration C and since only u and v made some moves from C to C_2 then we have: $\{w \in N(y) : p_w = y\} = \varnothing$ holds in C_2. Thus node y performs a *SingleNode* move and sets $p_y = null$. The edge (u, v) is now in state AlmostOff. 6

Note that in Figure 5.3.(d), edge $(9, 8)$ is in state AlmostOn. Figures 5.3.(e-g) represent the execution described in Lemma 6 leading to a configuration where the edge $(9, 8)$ is in state AlmostOff.

5.4.2 *The graph G_N and how to interpret a configuration into a binary integer*

In the following, we describe an execution corresponding to count from 0 to $2^N - 1$, where N is an arbitrary integer. This execution occurs in a graph denoted by G_N with $\Theta(N^2)$ nodes. G_N is composed by N sub-graphs, each of them representing a bit. The whole graph then represents an integer, coded from these N bits. G_N has 2 kind of nodes: the nodes represented by circles ($\bullet$ -nodes) and those represented by squares ($\square$ -nodes). The $\bullet$ -nodes are used to store bit values and hence an integer. The $\square$ -nodes are used to implement the "+1" operation as we count from 0 to $2^N - 1$.

We now formally describe the graph $G_N = (V_N, E_N)$:

1) $V_N = V_N^{\bullet} \cup V_N^{\square}$ where

$$V_N^{\bullet} = \bigcup_{0 \leqslant i < N} \{b(i,k) | k = 1, 2, 3, 4\}$$
$$V_N^{\square} = \bigcup_{0 \leqslant j < i < N} \{r_1(i,j), r_2(i,j)\}$$

2) $E_N = E_N^{\bullet} \cup E_N^{\square}$ where

$$E_N^{\bullet} = \bigcup_{0 \leqslant i < N} \{(b(i,k), b(i,k+1)) | k = 1, 2, 3\}$$
$$E_N^{\square} = \bigcup_{0 \leqslant j < i < N} \{(b(i,1), r_1(i,j)),\ (r_1(i,j), r_2(i,j)),\ (r_2(i,j), b(j,4))\}$$

Figure 5.5 gives a partial view of the graph G_N corresponding to the ith bit-block.

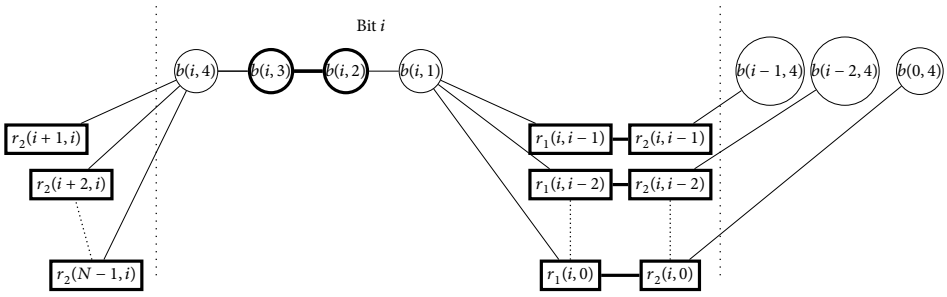


FIGURE 5.5
A partial view of graph G_N

Our exponential execution used the following underlying maximal matching $\mathcal{M}$:

$$\mathcal{M} = \{(b(i,2), b(i,3)) | 0 \leqslant i < N\} \cup \{(r_1(i,j), r_2(i,j)) | 0 \leqslant j < i < N\}$$

This maximal matching is encoded with the m-variables then we have:

$$\forall i, j \text{ with } 0 \leqslant j < i < N : m_{b(i,2)} = b(i,3), m_{b(i,3)} = b(i,2), m_{r_1(i,j)} = r_2(i,j) \text{ and}$$
$$m_{r_2(i,j)} = r_1(i,j)$$

The matching $\mathcal{M}$ is a $\frac{1}{2}$-approximation of the maximum matching and the algorithm ExpoMatch updates this approximation building $\mathcal{M}^+$, a $\frac{2}{3}$-approximation of the maximum matching. $\mathcal{M}^+$ is encoded with the p-variable and we also use this variable to encode the binary integer associated to a configuration.

EXAMPLE As an illustration, graph G_4 is shown in Figure 5.6. In this example, the bold edges are those belonging to the maximal matching $\mathcal{M}$ and arrows represent the local variable p of the algorithm ExpoMatch. A node having no outgoing arrow has its p-variable equal to *null*.

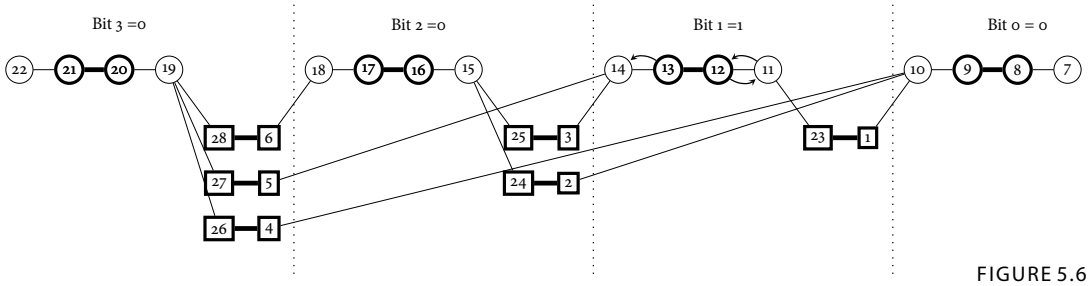


FIGURE 5.6
Graph G_4 encoding 0010

As we said, the •-nodes are used to encode the N bits. Each bit i is encoded with the local state of the 4 following nodes: $b(i,1), b(i,2), b(i,3), b(i,4)$. These nodes are

then named $b(i,k)$, for "*the k^{th} node of the bit i*". For instance, node 10 is the fourth node of the bit 0, thus node 10 is called $b(0,4)$. In the following, we will refer to these four nodes as the *i^{th} bit-block*. The binary value associated to a bit-block is computed accordingly to the p-values of each nodes in the bit-block. The following definition give this association:

DEFINITION 41. *(Bit-block encoding)*
*In graph G_N, nodes $\{b(i,1), b(i,2), b(i,3), b(i,4)\}$ are the i^{th} bit-block, for some $0 \leqslant i < N$. This bit-block encodes the value 1 (*resp. *0) if the edge $(b(i,2), b(i,3))$ is in state* On *(*resp. Off*) and if $\forall j$ with $0 \leqslant j < i$, $p_{r_1(i,j)} = p_{r_2(i,j)} = null$.*

Note that the value encoded by a bit-block is not always defined. But when all bit-blocks encode a bit in a given configuration, then we can associate a positive integer ω to this configuration.

DEFINITION 42. *(ω-configuration)*
Let ω be an integer such that $0 \leqslant \omega < 2^N$, a configuration C is said to be an ω-configuration *if for any integer $0 \leqslant i < N$, the i^{th} bit of ω is the value encoded by the i^{th} bit-block in C.*

Observe that all the p-values of the $\square$-nodes have to be *null* in any ω-configuration.

In Figure 5.6, all p-values of $\square$-nodes are *null*. Moreover, the edges $(9,8)$, $(17,16)$ and $(21,20)$ are in state Off while the edge $(13,12)$ is in state On. Thus, G_4 encodes the binary integer 0010 and so Figure 5.6 shows a 3-configuration.

5.4.3 *Identifiers in G_N*

In order to exhibit our execution counting from 0 to $2^N - 1$, we need to be able to switch edges between On and Off. This can be done executing the guarded rules of ExpoMatch. Since this algorithm uses identifiers, we need some properties on identifiers of nodes in G_N. The *Ident* function gives the identifier associated to a node in V_N. Recall that we assume each node has a unique identifier. These identifiers must satisfy the three following properties:

PROPERTY 1. *(Identifiers order in G_N)*
Let $b(i,k), b(i',k'), b(i,2)$ and $b(i,3)$ be nodes in $V_N^{\bullet}$, and $r_1(i,j)$ and $r_2(i,j)$ be nodes in $V_N^{\square}$. We have:

a) $Ident(b(i,k)) > Ident(b(i',k'))$ *if* $(i > i') \vee (i = i' \wedge k > k')$
b) $Ident(b(i,2)) < Ident(r_1(i,j))$
c) $Ident(b(i,3)) > Ident(r_2(j,i))$

We now show that in graph G_N, it exists an *Ident* function that satisfies Property 1. Indeed, the property holds for the following naming:
Let $c = |V_N^{\bullet}|$ and $s = \frac{|V_N^{\square}|}{2}$. There are c nodes of kind b, s nodes of kind r_1 and s nodes of kind r_2 as well.

- Nodes of kind r_2 are named from 1 to s
- Nodes of kind b are named from $s+1$ to $s+c$ such that:

$$\forall i, 0 \leqslant i < N, \forall k \in \{1,2,3,4\} : Ident(b(i,k)) = s + 4i + k$$

- Nodes of kind r_1 are named from $s+c+1$ to $s+c+s$

Figure 5.6 shows graph G_4 with such a naming ($c = 16$ and $s = 6$).

5.4.4 *Counting from 0 to $2^N - 1$*

We build an execution containing all ω-configurations with $0 \leqslant \omega < 2^N - 1$. To to this, we build an execution from an ω-configuration to the $(\omega+1)$-configuration using a '+1' operation. Thus we need to be able to switch bit from 0 to 1 and from 1 to 0. The main scheme is the following: let us consider a binary integer x. The '+1' operation consists in finding the rightmost 0 in x. Then all 1 at the right of this 0 have to switch to 0 and this 0 has to switch to 1 (if $x = x'011\ldots1$ then $x+1 = x'100\ldots0$). Then if 0 is the i^{th} bit of x, the i^{th} bit-block has to switch from 0 to 1 during the '+1'operation. And each j^{th} bit-block, with $0 \leqslant j < i$, has to switch from 1 to 0.

The switch of a bit-block from 0 to 1 only needs the •-nodes to perform moves (see Lemma 5). However, this is not the case when we want to switch a bit-block from 1 to 0. Indeed, we use some other nodes to help to perform the switch: the □-nodes.

Figures 5.6 to 5.11 show a part of an execution where we apply a '+1' operation twice. In Figure 5.6, the drawn graph encodes the integer (0010). Observe the edge $(b(0,2), b(0,3))$ from the 0^{th} bit-block is in state Off. We use Lemma 5 to go from the 0010-configuration represented in Figure 5.6 to the 0011-configuration represented in Figure 5.7. Figure 5.7 to 5.11 illustrate the transformation from the 0011-configuration to the 0100-configuration. Observe that Figures 5.8 to 5.10 do not encode any integer.

THEOREM 9. *Let ω be an integer such that $0 \leqslant \omega < 2^N - 1$. There exists a finite execution to transform an ω-configuration into an $(\omega+1)$-configuration.*

Proof of 9. Let C be an ω-configuration. Let ρ be the integer such that the $\rho - 1$ first bits of ω equal to 1 and the value of its ρ^{th} bit to 0. This implies that the ρ^{th} bit of $\omega + 1$ is the first bit equal to 1.

We distinguish two cases: $\rho = 0$ and $\rho > 0$.

In the case where $\rho = 0$, edge $(b(0,2), b(0,3))$ is in state Off by definition. Since the $0th$ bit of integer $\omega + 1$ is equal to 1, $(b(0,2), b(0,3))$ must be in state On in the $(\omega+1)$-configuration. By Property 1, we have $Ident(b(0,1)) < Ident(b(0,4))$. Moreover nodes $b(0,3)$ and $b(0,2)$ only have one single neighbor, so the hypotheses of Lemma 5 are satisfied. Thus, from Lemma 5, there exists an execution to switch edge $(b(0,2), b(0,3))$ from state Off to state On and in this execution, only nodes $b(0,1), b(0,2)$ and $b(0,3)$ make moves. At the end, the $0th$ bit has changed from 0 to 1 and the other did not change. We then have an $(\omega+1)$-configuration.

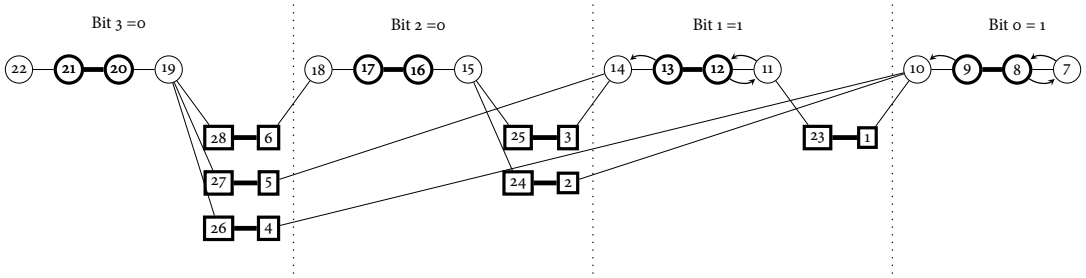


FIGURE 5.7
After turning on the 0th bit-block, G_4 encodes 0011.

In the case where $\rho > 0$, for every integer i from 0 to $\rho - 1$, edge $(b(i,2), b(i,3))$ is in state On and edge $(b(\rho,2), b(\rho,3))$ is in state Off.

We can execute the following sequence of moves to obtain the $(\omega{+}1)$-configuration:

a) We first consider the 3-augmenting path $(b(\rho,1), r_1(\rho,j), r_2(\rho,j), b(j,4))$ for any integer j, $0 \leqslant j < \rho$. We prove that the matched edge of this path is in state Off and that it satisfies the assumptions of Lemma 5. Then, we switch this edge from state Off to state On applying Lemma 5 (where the path (x,u,v,y) in this lemma corresponds to the path $(b(\rho,1), r_1(\rho,j), r_2(\rho,j), b(j,4))$).

Note that $\forall j, 0 \leqslant j < \rho$, node $r_1(\rho,j)$ (resp. $r_2(\rho,j)$) is adjacent to one single node $b(\rho,1)$ (resp. $b(j,4)$). As for any i, $0 \leqslant j < \rho$, the j^{th} bit-block encodes the value 1 in C, then $p_{b(j,4)} = null$ in C. In the same way, as the ρ^{th} bit-block encodes the value 0 in C, then $p_{b(\rho,1)} = null$ in C. As C is an ω-configuration, then $p_{r_1(\rho,j)} = null$ and $p_{r_2(\rho,j)} = null$. Thus the edge $(r_1(\rho,j), r_2(\rho,j))$ is in state Off in C.

Moreover, $Ident(b(j,4)) < Ident(b(\rho,1))$ by Property 1. Finally, in C, we have $\{w \in N(b(j,4)) : p_w = b(j,4)\} = \{b(j,3)\}$ since all neighbours of $b(j,4)$ but $b(j,3)$ are $\square$-nodes, and so they have their p-value equal to $null$ in C. We have $Ident(r_2(\rho,j)) < Ident(b(j,3))$ by Property 1, then $r_2(\rho,j) = min(\{w \in N(b(j,4)) : p_w = b(j,4)\} \cup \{r_2(\rho,j)\})$ and the hypotheses of Lemma 5 are satisfied. Thus from Lemma 5, we can exhibit an execution to switch edges $(r_1(\rho,j), r_2(\rho,j))$ from state Off to state On and where only nodes $r_1(\rho,j), r_2(\rho,j)$ and $b(j,4)$ make moves. The configuration shown in Figure 5.8 corresponds to this step.

b) Now, for each integer i, $0 \leqslant i < \rho$, edge $(b(i,2), b(i,3))$ is in state AlmostOn. $Ident(b(i,1)) < Ident(b(i,4))$ and $\{w \in N((b(i,1)) : p_w = (b(i,1)\} = \{b(i,2)\}$ so hypothesis of Lemma 6 hold. Thus from Lemma 6, an execution to switch edge $(b(j,2), b(j,3))$ from state AlmostOn to state AlmostOff is performed. The configuration shown in Figure 5.9 corresponds to this step.

c) Edge $(b(\rho,2), b(\rho,3))$ is still in state Off. Using the same argument of step (1), from Lemma 5, we can exhibit an execution to switch edges $(b(\rho,2), b(\rho,3))$ from state Off to state On. Figure 5.10 illustrates this step.

d) Now, for each integer j, $0 \leqslant j < \rho$, edge $(r_1(\rho,j), r_2(\rho,j))$ is now in state Al-

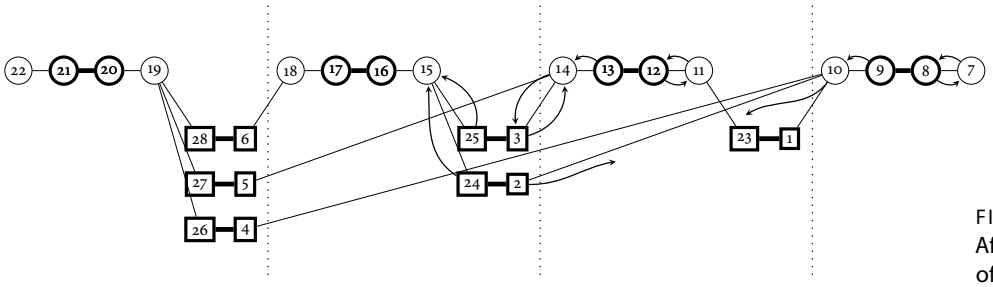


FIGURE 5.8
After activating the □ -nodes of the 3rd bit-block, G_4 does not encode any integer.

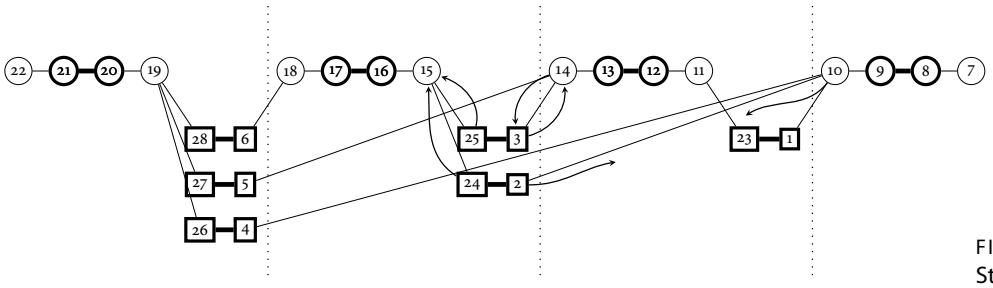


FIGURE 5.9
Starting to turn off the 0th and 1st bit-blocks.

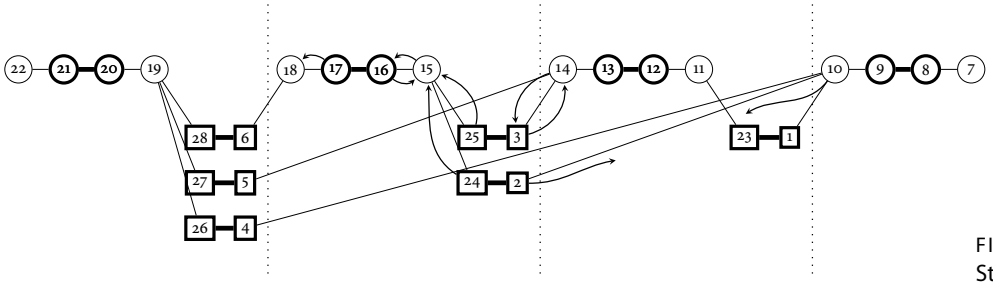


FIGURE 5.10
Starting to turn on the 3rd bit-block.

MOSTON. From Lemma 6, there exists an execution to switch edge $(r_1(\rho, j), r_2(\rho, j))$ from state ALMOSTON to state ALMOSTOFF. Figure 5.11 illustrates this step.

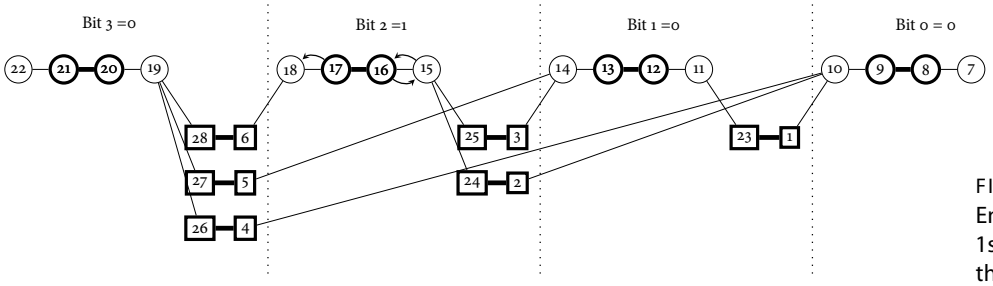


FIGURE 5.11
Ending to turn off the 0th and 1st bit-blocks and to turn on the 3rd bit-block. G_4 encodes 0100.

At the end of this execution, we obtain a configuration where the $\rho - 1$ first bits of ω are equal to 0 and he ρ^{th} bit is 1. Moreover, observe that all $\square$-nodes are in state AlmostOff or Off, thus they all have their p-value sets to *null*. We are then in an $(\omega + 1)$-configuration. 9

From now, we can construct an instance from which an execution having $\Omega(2^{\sqrt{n}})$ moves can be built.

COROLLARY 2. *Let n be the number of nodes. Algorithm ExpoMatch can stabilize after at most $\Omega(2^{\sqrt{n}})$ moves under the central daemon.*

Proof of 2. To prove the corollary, we can exhibit an execution of $\Omega(2^{\sqrt{n}})$ moves. Let N be an integer. The initial configuration is a 0-configuration in graph G_N.

We can build an execution that contains all the ω-configurations for every value ω, $0 \leqslant \omega < 2^N$. By applying Theorem 9, this execution can be split into 2^N parts corresponding to the execution from ω-configuration to $(\omega + 1)$-configuration, for $0 \leqslant \omega < 2^N$. Thus, this execution contains 2^N configurations. Since graph G_N has $O(N^2)$ vertices, this execution has $\Omega(2^{\sqrt{n}})$ configurations and the corollary holds. 2

5.5 THE NEW ALGORITHM POLYMATCH

We now follow up with the PolyMatch algorithm, based on the algorithm presented by Manne *et al.* [MMPT11], called ExpoMatch. As in ExpoMatch, PolyMatch assumes there exists an underlying maximal matching, called $\mathcal{M}$.

5.5.1 *Variables description*

Our algorithm has the same set of local variables as in ExpoMatch plus one additional boolean variable, called *end*. As in ExpoMatch, for a matched node u, the pointer p_u refers to a neighbor of u that u is trying to (re)match with, and pointers α_u and β_u refer to two candidates for a possible rematching with u. And again, s_u is a boolean variable that indicates if u has performed a successful rematching with its candidate. Finally, the new variable end_u is a boolean variable that indicates if *both* u and m_u have performed a successful rematching or not. For a single node x, only one pointer p_x and one boolean variable end_x are needed. p_x has the same purpose as the p-variable of a matched node. The *end*-variable of a single node allows the matched nodes to know whether it is *available* or not. A single node x is *available* for a matched node u if it is possible for x to eventually rematch with u, *i.e.*, $p_x = u \vee end_x =$*False* (see *BestRematch* predicate).

Algorithm 8 Functions used in the PolyMatch algorithm

function BestRematch(v)
  $a = Lowest\{x \in single(N(u)) \wedge (p_x = u \vee end_x = False)\}$
  $b = Lowest\{x \in single(N(u)) \setminus \{a\} \wedge (p_x = u \vee end_x = False)\}$
  return (a, b)

function AskFirst(u)
  if $\alpha_u \neq null \wedge \alpha_{m_u} \neq null \wedge 2 \leqslant Unique(\{\alpha_u, \beta_u, \alpha_{m_u}, \beta_{m_u}\}) \leqslant 4$ **then**
    if $(\alpha_u < \alpha_{m_u}) \vee (\alpha_u = \alpha_{m_u} \wedge \beta_u = null) \vee (\alpha_u = \alpha_{m_u} \wedge \beta_{m_u} \neq null \wedge u < m_u)$ **then**
      return α_u
  else
    return *null*

function AskSecond(u)
  if $AskFirst(m_u) \neq null$ **then**
    return $Lowest(\{\alpha_u, \beta_u\} \setminus \{\alpha_{m_u}\})$
  return *null*

$Unique(A)$ returns the number of unique elements in the multi-set A.
$Lowest(A)$ returns the node in A with the lowest identifier. If $A = \varnothing$, then $Lowest(A)$ returns *null*.

5.5.2 *Augmenting paths detection and exploitation*

AUGMENTING PATH DETECTION In this algorithm, a single node x is a candidate for a matched node u if it is not involved in another augmenting path that is fully exploited, *i.e.*, if $end_x = False \vee p_x = u$.

AUGMENTING PATH EXPLOITATION A 3-augmenting path is exploited in three phases. These phases are performed in a sequential way. Let us consider two nodes u and v such that $(u, v) \in \mathcal{M}$. Let us assume that u and v detects a 3-augmenting path.

1) The *First* node starts (same as in ExpoMatch): The *First* node, let say u, tries to rematch with its candidate. This phase is complete when $s_u = True$ and this indicate to the *Second* node (v) that the first phase is over.
2) The *Second* node continues: only when the first node succeeds will the second node attempt to rematch with one of its candidates. This phase is complete when $end_v = True$ and this indicate to the v's neighbours that the second phase is over.
3) All nodes in the path set their *end* variable to *True*: the *end* value of v is propagated in the path. The goal of this phase is to write *True* in the *end* variables of the two single nodes in the path in order to make them unavailable for other

Algorithm 9 PolyMatch algorithm

▷**ResetEnd** ▷ Rules for each node u in **single**(**V**)
if $p_u = null \wedge end_u = True$ **then**
$end_u := False$

▷**UpdateP**
if $(p_u = null \wedge \{w \in matched(N(u)) \mid p_w = u\} \neq \emptyset) \vee (p_u \notin (matched(N(u)) \cup \{null\})) \vee (p_u \neq null \wedge p_{p_u} \neq u)$ **then**
$p_u := Lowest\{w \in N(u) \mid p_w = u\}$
$end_u := False$

▷**UpdateEnd**
if $(p_u \in matched(N(u)) \wedge (p_{p_u} = u) \wedge (end_u \neq end_{p_u})$ **then**
$end_u := end_{p_u}$

▷**Update** ▷ Rules for each node u in **matched**(**V**)
if $(\alpha_u > \beta_u) \vee (\alpha_u, \beta_u \notin (single(N(u)) \cup \{null\})) \vee (\alpha_u = \beta_u \wedge \alpha_u \neq null) \vee p_u \notin (single(N(u)) \cup \{null\}) \vee ((\alpha_u, \beta_u) \neq BestRematch(u) \wedge (p_u = null \vee (p_{p_u} \neq u \wedge end_{p_u} = True)))$ **then**
$(\alpha_u, \beta_u) := BestRematch(u)$
$(p_u, s_u, end_u) := (null, False, False)$

▷**MatchFirst**
if $(AskFirst(u) \neq null) \wedge [\ p_u \neq AskFirst(u) \vee s_u \neq (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_{m_u} \in \{AskSecond(m_u), null\}) \vee end_u \neq (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = AskSecond(m_u) \wedge end_{m_u})\ ]$ **then**
$end_u := (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = AskSecond(m_u) \wedge end_{m_u})$
$s_u := (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge (p_{m_u} \in \{AskSecond(m_u), null\})$
$p_u := AskFirst(u)$

▷**MatchSecond**
if $AskSecond(u) \neq null) \wedge (s_{m_u} = True) \wedge [\ p_u \neq AskSecond(u) \vee end_u \neq (p_u = AskSecond(u) \wedge p_{p_u} = u \wedge p_{m_u} = AskFirst(m_u)) \vee s_u \neq end_u\ ]$ **then**
$end_u := (p_u = AskSecond(u) \wedge p_{p_u} = u \wedge p_{m_u} = AskFirst(m_u))$
$s_u := end_u$
$p_u := AskSecond(u)$

▷**ResetMatch**
if AskFirst(u) = AskSecond(u) = $null \wedge (p_u, s_u) \neq (null, false)$ **then**
$(p_u, s_u) := (null, false)$

married nodes. Indeed, the *end* variable is used to compute the candidates of a matched node.

The scenario for an augmenting path exploitation when everything goes well is given in the following. Node u starts trying to rematch with x performing a *MatchFirst* move and $p_u := x$. If x accepts the proposition, performing an *UpdateP* move and $p_x := u$, then u will inform v of this first phase success, once again by performing a *MatchFirst* move and $s_u :=$*True*. Observe that at this point, x cannot change its p-value since $p_{p_x} = x$. Finally, node v tries to rematch with y, performing a *MatchSecond* move and $p_v := y$. If y accepts the proposition, performing an *UpdateP* move and $p_y := v$, then v will inform u of this final success, by performing a *MatchSecond* move again and $end_v :=$*True*. This complete the second phase. From then, all nodes in this 3-augmenting path will set there *end*-variable to *True*: u by performing a last *MatchFirst* move, and x and y by performing an *UpdateEnd* move. From this point, non of these nodes x, u, v, or y will ever be eligible for any move again. Moreover, once single nodes have their *end*-variables set to *True*, they are not available anymore for any other matched nodes.

5.5.3 *Rules description*

There are four rules for matched nodes. As in ExpoMatch, the *Update* rule allows a matched node to update its α and β variables, using the *BestRematch* predicate. Then, predicates *AskFirst* and *AskSecond* are used to define the role the node will have in the 3-augmenting path exploitation. If the node is *First* (resp. *Second*), then it will execute *MatchFirst* (resp. *MatchSecond*) for this 3-augmenting path exploitation. The *ResetMatch* rule is performed to reset bad initialization and also to reset an augmenting path exploitation that did not terminate.

The *MatchFirst* rule is used by the node when it is *First*. Let u be this node. The rule is performed three times in a usual path exploitation:

1) The first time this rule is performed, u seduces its candidate setting (end_u, s_u, p_u) to $(False, False, AskFirst(u))$.
2) Then this rule is performed a second time after the u's candidate has accepted the u's proposition, *i.e.*, when $AskFirst(u)$ has set its p-variable to u. So the second *MatchFirst* execution sets (end_u, s_u, p_u) to $(False, True, AskFirst(u))$. Now, variable s_u is equal to $True$, allowing node m_u that is *Second* to seduce its own candidate.
3) Finally, the *MatchFirst* rule is performed a third time when m_u completed is own rematch, *i.e.*, when $end_{m_u} = True$. Observe that when there is no bad information due to some bad initializations, then $end_{m_u} = True$ means $p_{m_u} = AskSecond(m_u) \wedge p_{p_{m_u}} = m_u$ (see the third line of the *MatchSecond* rule). So this third *MatchFirst* execution sets (end_u, s_u, p_u) to $(True, True, AskFirst(u))$, meaning that the 3-augmenting path has been fully exploited.

In the *MatchFirst* rule, observe that we make the s_u affectation before the p_u affectation, because the s_u value must be computed accordingly to the value of

p_u before activating u. Indeed, when u executes *MatchFirst* for the first time, it allows to set p_u from $\perp$ to $AskFirst(u)$ while s_u remains *False*. Then when u executes *MatchFirst* for the second time, s_u is set from *False* to *True* while p_u remains equal to $AskFirst(u)$. For the same argument, we make the end_u affectation before the s_u affectation. Thus, the "normal" values sequence for (p_u, s_u, end_u) is: $((\perp, False, False), (AskFirst(u), False, False), (AskFirst(u), True, False), (AskFirst(u), True, True))$.

The *MatchSecond* rule is used by the node when it is *Second*. This rule is performed only twice in a usual path exploitation. For the first execution, u has to wait for m_u to set its s_{m_u} to *True*. Then u can perform *MatchSecond* and update its p-variable to $AskSecond(u)$. When the u's candidate has accepted his proposition, u can perform *MatchSecond* for the second time, setting s_u and end_u to *True*. As in the *MatchFirst* rule, we set the *end* and s affectations before the p affectation.

There are three rules for single nodes. The *ResetEnd* rule is used to reset bad initializations. In the *UpdateP* rule, the node updates its p-value according to the propositions done by neighbouring matched nodes. If there is no proposition, the node sets its p-value to *null*. Otherwise, p is set to the minimum identifier among all proposals. Afterwards, the p-value can only change when the proposition is cancelled. When a single node u has accepted a proposition, its end value should be equal to the end value of p_u. The *UpdateEnd* rule is used for this purpose.

5.5.4 *Execution examples*

We give two different executions of algorithm PolyMatch under the adversarial distributed daemon. The first execution points out the main differences between our algorithm PolyMatch and algorithm ExpoMatch. In the second execution, we focus on the *end* variable role for the exploited path process.

MAIN DIFFERENCE BETWEEN POLYMATCH AND EXPOMATCH ALGORITHMS We will consider the same example in Section 5.3.3: we assume that we are in configuration drawn in Figure 5.3.(d). We assume that all *end-values* equals to *False*. We also assume that all α*-values* and β*-values* are defined as follows: $(\alpha_8, \beta_8) = (7, null)$, $(\alpha_9, \beta_9) = (10, null)$ and $(\alpha_{24}, \beta_{24}) = (15, null)$ and $(\alpha_2, \beta_2) = (10, null)$. At this moment, we assume that the two 3-augmenting paths are partially exploited : $p_2 = p_9 = 10$, $p_{10} = 2$, $p_{15} \neq 24$, $p_{15} \neq null$, $p_8 = 7$, and $p_7 = 8$.

Let focus on node 24. Node 24 considers that it does not belong to a 3-augmenting path because $end_{15} = True$ means that node 15 is not rematched. Thus, it is eligible to execute a *Update* move.

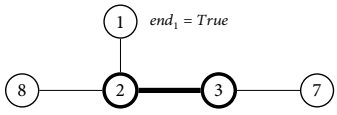
In our algorithm, even after node 10 has chosen node 2, node 9 still waits for an acceptation of node 10, and will do so while end_{10} remains *False* except for node 15. However, at this point, in Manne *et al.* algorithm, node 9 can destroy the augmenting-path construction. This is the main difference that allows our algorithm to prevent from exponential executions.

So, at this point there is a binary choice for node 9: destroy or not its augmenting-path construction. In Manne *et al.* algorithm, the choice is to destroy, thus the destruction of a partially exploited augmenting-path can be done while no fully exploited augmenting path has been built. Moreover, for one fully exploited augmented path, we can exhibit some executions where we destroy a sub-exponential number of exploited augmented-path (see Section 5.4). In our algorithm, we do the other choice which is: do not destroy while there is still hope to exploit the augmenting path. So, if node 9 breaks a partially exploited augmenting path, then node 10 belongs to a fully exploited augmenting-path. Thus, this destruction implies one 3-augmenting path has been fully exploited, and thus the matching size has been increased by 1.

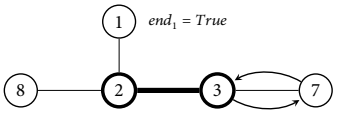
This difference is implemented in the algorithm through the *BestRematch* predicate. The condition $p_x = null$ in Manne *et al.* algorithm has been replaced by the condition end_x =*False* in our algorithm. Then, in our algorithm, *BestRematch*(9) remains constant when node 10 chooses node 2, while it does not in Manne *et al.* algorithm, making node 9 eligible for *Update*.

How to handle the end-variable.

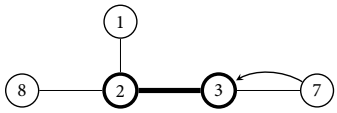
Second, we consider the first execution in order to illustrate the rule of local *end*-variable. Figure 5.12.(a) shows the initial state of the execution. The underlying maximal matching contains one edge (2, 3). Then nodes 2, 3 are *matched* nodes, and nodes 1, 7, and 8 are *single* nodes. At the beginning, there are two 3-augmenting paths: (1, 2, 3, 7) and (8, 2, 3, 7).



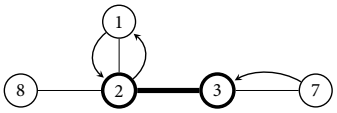
(a) Initial configuration.



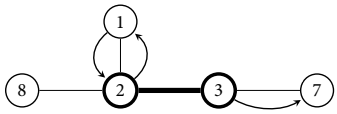
(b) 3 executes *MatchFirst*, then 7 executes *UpdateP* and chooses 3.



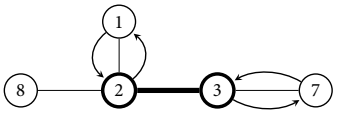
(c) 1 executes *ResetEnd*, then 2 executes *Update* and becomes *First*. Finally, 3 executes *ResetMatch*.



(d) 2 executes *MatchFirst*, then 1 executes *UpdateP* and accepts the proposition of 2. Finally, 2 executes *MatchFirst* (s_2:=*True*).



(e) In parallel 7 and 3 execute *UpdateP* and *MatchSecond* respectively.



(f) 7 executes *UpdateP*, then 3 executes *MatchSecond*, then the *True* value of end_3 is propagated in the path (1, 2, 3, 7).

FIGURE 5.12
An execution of Algorithm PolyMatch (Only the *True* value of the *end*-variables are given)

The initial configuration (Figure 5.12.(a)):

In the initial configuration, we assume that all α-*values* and β-*values* are defined as follows: $(\alpha_2, \beta_2) = (8, null)$, and $(\alpha_3, \beta_3) = (7, null)$. We also assume all *s-values* are well defined (all other *s*-values are *False*) whereas all *end*-values are *False* but end_1 that is *True*. At this moment, node 2 considers that since end_1 =*True*, node 1 already belongs to a fully exploited 3-augmenting path: $BestRematch(2) = (8, null)$.

The 3-augmenting path is (7, 3, 2, 8). Node 2 considers that node 1 is not available because end_1 =*True*. Since $2 \leqslant Unique(\{\alpha_2, \beta_2, \alpha_3, \beta_3\}) \leqslant 4$, nodes 2 and 3 detect a 3-augmenting path and start to exploit it. Since node 3 is *First* ($AskFirst(3) = 7$ and $AskFirst(2) = null$), node 3 may execute a *MatchFirst* move. Let us assume it does.

The 3-augmenting path exploitation starts (Figure 5.12.(b)):

Node 3 executes here a *MatchFirst* move and points to node 7. Since node 3 is pointing to node 7, node 7 is the only activable node among all nodes except node 1. Node 7 points to node 3 by executing a *UpdateP* move.

Let us focus on node 1. Its *end*-value is not well defined since end_1 =*True* while node 1 does not belong to a fully exploited augmenting path. Thus, node 1 is eligible for $ResetEnd$ rule. Let us assume it makes this move. After this move, we have end_1 =*False*. This implies that $BestRematch(2) = (1, 8)$ and thus $(\alpha_2, \beta_2) = (8, null) \neq BestRematch(2)$. So, only node 2 is activable, and is eligible for $Update$ rule. Thus, after this mode, node 2 is *First*. This implies that node 3 is *Second*, and it is eligible for $ResetMatch$ because $AskSecond(3) \neq null \wedge p_3 \neq null \wedge s_2$ =*False*. Let us assume it does it.

A second 3-augmenting path exploitation starts (Figure 5.12.(d)):

Let us consider node 2. It is *First* and it can execute a *MatchFirst* rule. After this activation, it sets $p_2 = 1$ and $s_2 = end_2$ =*False*. Now, node 1 accepts the node 2 proposition by executing a *UpdateP* move. After this activation, node 1 points to node 2 ($p_1 = 2$). Now, node 2 is eligible for executing a *MatchFirst* rule. It sets $p_2 = 1$ and s_2 = *True*. This implies that node 3 becomes eligible for *MatchSecond*.

In the configuration shown in Figure 5.12.(e), node 3 can propose to node 7 with a *MatchSecond*. Note that node 7 is also eligible for $UpdateP$ since $p_3 \neq 7$. Let us assume these two nodes do the move in parallel. Figure 5.12.(e) shows the configuration obtained after theses moves: $p_3 = 7$, $p_7 = null$. Note that after these activations, we have s_3 = *False* since, *before* these activations, the p-values of nodes 3 and 7 are not as follow: $p_3 = 7$ and $p_7 = 3$. This kind of transitions, where a matched node proposition is performed in parallel with a single node abandonment, is the reason why we make the *s*-affectation, then the p-affectation in the *MatchFirst* rule. This trick allows to obtain after a *MatchFirst* rule: s_u = *True* implies $p_{p_u} = u$. Finally, observe at this step that node 3 still waits for an answer of node 7.

The path $(1, 2, 3, 7)$ *becomes fully exploited (Figure 5.12.(f)):*

Now, node 7 can choose 3 by executing *UpdateP*. Assume that it does. Since $end_3 \neq (p_3 = 7 \wedge p_3 = AskSecond(3) \wedge p_2 = AskFirst(2))$, node 3 is eligible for a *MatchSecond* rule to set end_3 to *True* and then to make the other nodes aware that the path is fully exploited. Assume node 3 executes a *MatchSecond* move. This will cause node 7 (resp. 2) to execute an *UpdateEnd* move (resp. a *MatchFirst* move) and sets end_7 =*True* (resp. end_2 =*True*). Now, it is the turn to node 1 to execute an *UpdateEnd* move. As the *end*-value of nodes 1, 2, 3, and 7 are equal to *True*, the 3-augmenting path is fully exploited. The system has reached a stable configuration (see Figure 5.12.(f)). Thus, the size of the matching is increasing by one and there is no 3-augmenting path left.

Now, we present the proof of our algorithm.

5.6 CORRECTNESS PROOF

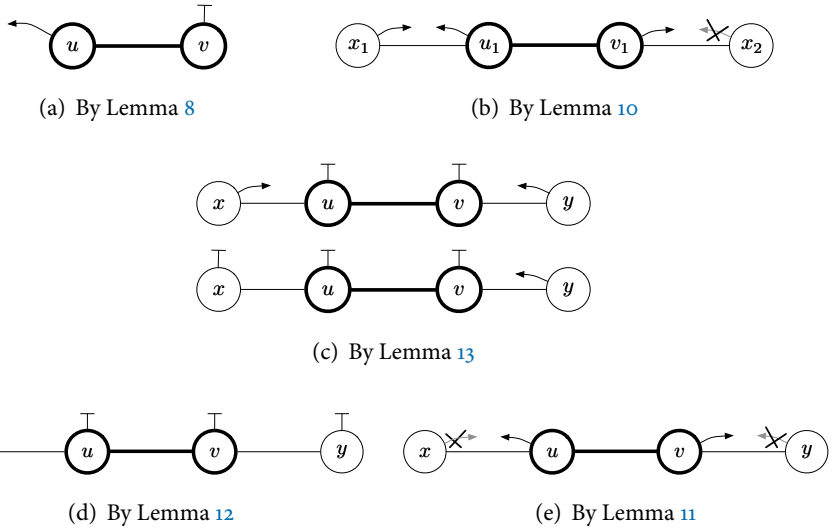
A natural way to prove the correction of PolyMatch algorithm could have been to follow the approach below. We consider a stable configuration C in PolyMatch and we prove C is also stable in the Manne *et al.* algorithm. As we use the exact same variables but the *end*-variable and because the matching is only defined on the common variables, the correctness follows from Manne *et al.* paper. Moreover, we can easily show that if C is stable in PolyMatch, then no rule from the Manne *et al.* algorithm but the *Update* rule can be performed in C. Unfortunately, it is not straightforward to prove that the *Update* rule from Manne *et al.* algorithm cannot be executed in C. Indeed, our *Update* rule is more difficult to execute than the one of Manne *et al.* in the sens that some possible *Update* in Manne *et al.* are not possible in our algorithm. By the way, this is why our algorithm has a better time complexity since the number of partially exploited augmented path destruction in our algorithm is smaller than in the Manne *et al.* algorithm. In particular, we have to prove that in a stable configuration, for any matched node, if $p_u \neq null$, then $end_{p_u} = True$. To prove that, we need Lemmas 7, 8, 9, 10, 11 and a part of the proof from Theorem 10. Observe that from these results, the correctness is straightforward without using the Manne *et al.* proof.

We first introduce some notations. A matched node u is said to be *First* if $AskFirt(u) \neq null$. In the same way, u is *Second* if $AskSecond(u) \neq null$. Let $Ask : V \rightarrow V \cup \{null\}$ be a function where $Ask(u) = AskFirst(u)$ if $AskFirst(u) \neq null$, otherwise $Ask(u) = AskSecond(u)$. We will say a node makes a *match* rule if it performs a *MatchFirst* or *MatchSecond* rule.

Recall that the set of edges built by our algorithm PolyMatch is

$$\mathcal{M}^+ = \{(u, v) \in \mathcal{M} : p_u = p_v = null\} \cup \{(a, b) \in E \setminus \mathcal{M} : p_a = b \wedge p_b = a\}$$

For the correctness part of the proof, we prove that in a stable configuration, $\mathcal{M}^+$ is a 2/3-approximation of a maximum matching on graph G. To do that we demonstrate there is no 3-augmenting path on $(G, \mathcal{M}^+)$. In particular we prove that for any edge $(u, v) \in \mathcal{M}$, we have either $p_u = p_v = null$, or u and v have two distinct single neighbours they are rematched with, *i.e.*, $\exists x \in single(N(u)), \exists y \in single(N(v))$ with $x \neq y$ such that $(p_x = u) \wedge (p_u = x) \wedge (p_y = v) \wedge (p_v = y)$. In order to prove that, we show every other case for (u, v) is impossible. Main studied cases are shown in Figure 5.13. Finally, we prove that if $p_u = p_v = null$ then (u, v) does not belong to a 3-augmenting-path on $(G, \mathcal{M}^+)$.



(a) By Lemma 8 (b) By Lemma 10

(c) By Lemma 13

(d) By Lemma 12 (e) By Lemma 11

FIGURE 5.13
Impossible situations in a stable configuration.

LEMMA 7. *In any stable configuration, we have the following properties:*

- $\forall u \in matched(V) : p_u = Ask(u)$;
- $\forall x \in single(V)$: *if* $p_x = u$ *with* $u \neq null$, *then* $u \in matched(N(x)) \wedge p_u = x \wedge end_u = end_x$.

Proof of 7. First, we will prove the first property. We consider the case where $AskFirst(u) \neq null$. We have $p_u = AskFirst(u)$, otherwise node u can execute rule *AskFirst*. We can apply the same result for the case where $AskSecond(u) \neq null$. Finally, we consider the case where $AskFirst(u) = AskSecond(u) = null$. If $p_u \neq null$, then node u can execute rule *ResetMatch* which yields the contradiction. Thus, $p_u = null$.

Second, we consider a stable configuration C where $p_x = u$, with $u \neq null$. $u \in matched(N(x))$, otherwise x is eligible for an *UpdateP* rule. Now there are two cases: $p_u = x$ and $p_u \neq x$. If $p_u \neq x$, this means that $p_{p_x} \neq x$. Thus, x is eligible for rule *UpdateP*, and this yields to a contradiction with the fact that C is stable. Finally, we have $end_u = end_x$, otherwise x is eligible for rule *UpdateEnd*. 7

LEMMA 8. *Let (u, v) be an edge in $\mathcal{M}$. Let C be a configuration. If $p_u \neq null \wedge p_v = null$ holds in C (see Figure 5.13.(a)), then C is not stable.*

Proof of 8. By contraction. We assume C is stable. From Lemma 7, we have $p_u = Ask(u) \neq null$ and $p_v = Ask(v)$. So, by definition of predicates *AskFirst* and *AskSecond*, $Ask(u) = x \neq null$ implies that $Ask(v) \neq null$. This contradicts that fact that $p_v = Ask(v) = null$. □

LEMMA 9. *Let (x, u, v, y) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a stable configuration. In C, if $p_x = u$, $p_u = x$, $p_v = y$ and $p_y = u$, then $end_x = end_u = end_v = end_y = True$.*

Proof of 9. From Lemma 7, $p_u = Ask(u)$ (resp. $p_v = Ask(v)$) thus $Ask(u) \neq null$ and $Ask(v) \neq null$. W.l.o.g, we can assume that $AskFirst(u) \neq null$. We have $s_u = True$, otherwise u can execute *MatchFirst* rule. Now, as $s_u = True$, we must have $end_v = True$, otherwise v can execute *MatchSecond* rule. As $s_u = end_v = True$, we must have $end_u = True$, otherwise u can execute *MatchFirst* rule. From Lemma 7, we can deduce that $end_x = end_u = end_v = end_y = True$ and this concludes the proof. □

LEMMA 10. *Let (x_1, u_1, v_1, x_2) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a configuration. If $p_{x_1} = u_1 \wedge p_{u_1} = x_1 \wedge p_{v_1} = x_2 \wedge p_{x_2} \neq v_1$ holds in C (see Figure 5.13.(b)), then C is not stable.*

Proof of 10. By contraction. We assume C is stable. From Lemma 7, $Ask(u_1) = x_1$ and $Ask(v_1) = x_2$.

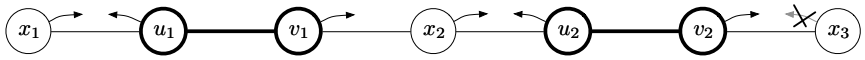
First we assume that $AskSecond(u_1) = x_1$ and $AskFirst(v_1) = x_2$. The local variable s_{v_1} is *False*, otherwise v_1 would be eligible for executing the *MatchFirst* rule. Since $AskSecond(u_1) \neq null \wedge p_{u_1} \neq null \wedge s_{v_1} = False$, this implies that u_1 is eligible for the *ResetMatch* rule which is a contradiction.

Second, we assume that $AskFirst(u_1) = x_1$ and $AskSecond(v_1) = x_2$. We have $s_{u_1} = True$, otherwise u_1 can execute the *MatchFirst* rule. This implies that $end_{v_1} = False$, otherwise v_1 can execute the *MatchSecond* rule. As $end_{v_1} = False$, then $end_{u_1} = False$, otherwise u_1 can execute the *MatchFirst* rule. From Lemma 7, $end_{x_1} = end_{u_1} = end_{v_1} = False$. Since $Ask(v_1) = x_2$, we have $x_2 \in \{\alpha_{v_1}, \beta_{v_1}\}$. Let us assume $end_{x_2} = True$. Then $x_2 \notin BestRematch(v_1)$ and then v_1 is elligible for an *Update*. Thus $end_{x_2} = False$.

Therefore, C is a configuration such that u_1 is *First* and v_1 is *Second* with $end_{x_1} = end_{u_1} = end_{v_1} = end_{x_2} = False$. Now we are going to show there exists another augmenting path (x_2, u_2, v_2, x_3) with $end_{x_2} = end_{u_2} = end_{v_2} = end_{x_3} = False$ and $p_{u_2} = x_2$, $p_{x_2} = u_2$, $p_{v_2} = x_3$ and $p_{x_3} \neq v_2$ such that u_2 is *First* and v_2 is *Second* (see Figure 5.14).

$p_{x_2} \neq null$ otherwise x_2 is eligible for an *UpdateP* rule. Thus there exists a vertex $u_2 \neq v_1$ such that $p_{x_2} = u_2$. From Lemma 7, $u_2 \in matched(N(x_2))$ and $p_{u_2} = x_2$. Therefore, there exists a node $v_2 = m_{u_2}$. From Lemma 8, we can deduce that $p_{v_2} \neq null$ and there exists a node x_3 such that $p_{v_2} = x_3$. $x_3 \in single(N(v_2))$

FIGURE 5.14
A chain of 3-augmenting paths.



otherwise x_2 is eligible for an *Update* rule. Finally, if $p_{x_3} = v_2$, then Lemma 9 implies that $end_{x_2} = end_{a_2} = end_{b_2} = end_{x_3} = True$. This yields to the contradiction with the fact $end_{x_2} = False$. So, we have $p_{x_3} \neq v_2$.

We can then conclude that (x_2, u_2, v_2, x_3) is a 3-augmenting path such that $p_{x_2} = u_2 \wedge p_{u_2} = x_2 \wedge p_{v_2} = x_3 \wedge p_{x_3} \neq v_2$. This augmenting path has the exact same properties than the first considered augmenting path (x_1, u_1, v_1, x_2) and in particular u_1 is First.

Now we can continue the construction in the same way. Therefore, for C to be stable, it has to exist a chain of 3-augmenting paths $(x_1, u_1, v_1, x_2, u_2, v_2, x_3, \ldots, x_i, u_i, v_i, x_{i+1}, \ldots)$ where $\forall i \geqslant 1 : (x_i, u_i, v_i, x_{i+1})$ is a 3-augmenting path with $p_{x_i} = u_i \wedge p_{u_i} = x_i \wedge p_{v_i} = x_{i+1} \wedge p_{x_{i+1}} = v_{i+1}$ and u_i is First. Thus, $x_1 < x_2 < \ldots < x_i < \ldots$ since the u_i will always be First. Since the graph is finite some x_k must be equal to some x_ℓ with $\ell \neq k$ which contradicts the fact that the identifier' sequence is strictly increasing. □10

LEMMA 11. *Let (x, u, v, y) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a configuration. If $p_u = x \wedge p_x \neq u \wedge p_v = y \wedge p_y \neq v$ holds in C (see Figure 5.13.(e)), then C is not stable.*

Proof of 11. By contradiction, assume C is stable. From Lemma 7, $Ask(u) = x$. Assume to begin that $AskFirst(u) \neq null$. Because $p_{p_u} \neq u$ we have $s_u = False$, otherwise u is eligible for *MatchFirst*. Since $AskSecond(v) \neq null$ and $s_{m_v} = s_u = False$ then v can apply the *ResetMatch* rule which yields a contradiction. Therefore assume that $AskSecond(u) \neq null$. The situation is symmetric (because now $AskFirst(v) \neq null$) and therefore we get the same contradiction as before. □11

LEMMA 12. *Let (x, u, v, y) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a configuration. If $p_y = p_u = p_v = p_y = null$ holds in C (see Figure 5.13.(d)), then C is not stable.*

Proof of 12. By contradiction, assume C is stable. $end_x = False$ (resp. $end_y = False$), otherwise x (resp. y) is eligible for a *ResetMatch*. $(\alpha_u, \beta_u) = BestRematch(u)$ (resp. $(\alpha_v, \beta_v) = BestRematch(v)$), otherwise u (resp. v) is eligible for an *Update*. Thus, there is at least an available single node for u and v and so $Ask(u) \neq null$ and $Ask(v) \neq null$. Then, this contradicts the fact that $Ask(u) = null$ (see Lemma 7). □12

THEOREM 10. *In a stable configuration we have, $\forall (u, v) \in \mathcal{M}$:*

- $p_u = p_v = null$ *or*
- $\exists x \in single(N(u)), \exists y \in single(N(v))$ *with* $x \neq y$ *such that* $p_x = u \wedge p_u = x \wedge p_y = v \wedge p_v = y$.

Proof of 10. We will prove that all cases but these two are not possible in a stable configuration. First, Lemma 8 says the configuration cannot be stable if exactly one of p_u or p_v is not *null*.

Second, assume that $p_u \neq null \wedge p_v \neq null$. Let $p_u = x$ and $p_v = y$. Observe that $x \in single(N(u))$ (resp. $y \in single(N(v))$), otherwise u (resp. v) is eligible for *Update*.

Case $x \neq y$: If $p_x \neq u$ and $p_y \neq v$ then Lemma 11 says the configuration cannot be stable. If $p_x = u$ and $p_y \neq v$ then Lemma 10 says the configuration cannot be stable. Thus, the only remaining possibility when $p_u \neq null$ and $p_v \neq null$ is: $p_x = p_u$ and $p_y = v$.

Case $x = y$: $Ask(u)$ (resp. $Ask(v) \neq null$), otherwise u (resp. v) is eligible for a *ResetMatch*. W.l.o.g. let us assume that u is First. $x = AskFirst(u)$ (resp. x=AskSecond(v)), otherwise u (resp. v) is eligible for *MatchFirst* (resp. *MatchSecond*). Thus $AskFirst(u) = AskSecond(v)$ which is impossible according to these two predicates. □10

LEMMA 13. *Let x be a single node. In a stable configuration, if $p_x = u, u \neq null$ then it exists a 3-augmenting path (x, u, v, y) on $(G, \mathcal{M})$ such that $p_x = u \wedge p_u = x \wedge p_v = y \wedge p_y = v$.*

Proof of 13. By lemma 7, if $p_x = u$ with $u \neq null$ then $u \in matched(N(x))$ and $p_u = x$. Since $p_u \neq null$, by Theorem 10 the result holds. □13

Observe that according to this Lemma, cases from Figure 5.13.(c) are impossible.

Thus, in a stable configuration, for all edges $(u, v) \in \mathcal{M}$, if $p_u = p_v = null$ then (u, v) does not belong to a 3-augmenting-path on $(G, \mathcal{M}^+)$. In other words, we obtain:

COROLLARY 3. *In a stable configuration, there is no 3-augmenting path on $(G, \mathcal{M}^+)$ left.*

5.7 CONVERGENCE PROOF

This section is devoted to a sketch of the convergence proof. In the following, μ will denote the number of matched nodes and σ the number of single nodes.

The first step consists in proving that the values of s and *end* represent the different phases of the path exploitation. Recall that $s_u = True$ means $p_{p_u} = u$. Moreover $end_u = True$ means that the path is fully exploited. We can easily prove that after one activation of a matched node u, $s_u = True$ implies $p_{p_u} = u$:

15 LEMMA 1. *Let u be a matched node. Consider an execution $\mathcal{E}$ starting after u executed some rule. Let C be any configuration in $\mathcal{E}$. In C, if s_u =True then $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$.*

However, a bad initialization of end_{m_u} to *True* can induce u to wrongly write *True* in end_u. But this can appear only once and thus, the second times u writes *True* in end_u means that a 3-augmenting path involving u has been fully exploited.

11 THEOREM 1. *In any execution, a matched node u can write* end_u :=*True at most twice.*

We now count the number of destruction of partially exploited augmenting paths. Recall that in Manne *et al.* algorithm, for one fully exploited augmenting path, it is possible to destroy a sub-exponential number of partially exploited ones.

In our algorithm, observe that for a path destruction, the set of single neighbors that are candidates for a matched edge has to change and this change can only occur when a single node changes its *end*-value. Such a change induces a path destruction if a matched node takes into account this modification by applying an *Update* rule. So, we first count the number of time a single node can change its *end*-value (Lemma 25) and then we deduce the number of time a matched node can execute *Update* (Corollary 5). Finally, we conclude we destroy at most $O(n^2)(= O(\Delta(\sigma+\mu)))$ partially exploited augmenting path.

The rest of the proof consists in counting the number of moves that can be performed between two *Update*, allowing us to conclude the proof (Theorem 12).

In the following, we detailed point by point the idea behind each result cited above.

Since single nodes just follow orders from their neighboring matched nodes, we can count the number of times single nodes can change the value of their *end* variable. There are σ possible modifications due to bad initializations. A matched node u can write *True* twice in end_u, so end_u can be *True* during 3 distinct sub-executions. As a single node x copies the *end*-value of the matched node it points to ($p_x = u$), then a single node can change its *end*-value at most 3 times as well. And we obtain 6μ modifications.

25 LEMMA 2. *In any execution, the number of transitions where a single node changes the value of its end variables (from True to False or from False to True) is at most* $\sigma + 6\mu$ *times.*

We count the maximal number of *Update* rule that can be performed in any execution. To do that, we observe that the first line of the *Update* guard can be *True* at most once in an execution (Lemma 16). Then we prove for the second line of the guard to be *True*, a single node has to change its *end* value. Thus, for each single node modification of the *end*−value, at most all matched neighbors of this single node can perform an *Update* rule.

5 COROLLARY 1. *Matched nodes can execute at most* $\Delta(\sigma+6\mu)+\mu$ *times the* Update *rule.*

Third, we consider two particular matched nodes u and v and an execution with no *Update* rule performed by these two nodes. Then we count the maximal number

of moves performed by these two nodes in this execution. The idea is that in such an execution, the α and β values of u and v remain constant. Thus, in these small executions, u and v detect at most one augmenting path and perform at most one rematch attempt. We obtain that the maximal number of moves of u and v in these small executions is 12. By the previous remark and Corollary 5, we obtain:

12 THEOREM 2. *In any execution, matched nodes can execute at most* $12\Delta(\sigma + 6\mu) + 18\mu$ *rules.*

Finally, we count the maximal number of moves that single nodes can perform, counting rule by rule. The *ResetEnd* is done at most once. The number of *UpdateEnd* is bounded by the number of times single nodes can change their *end*-value, so it is at most $\sigma + 6\mu$. Finally, *UpdateP* is counted as follows: between two consecutive *UpdateP* executed by a single node x, a matched node has to make a move. The total number of executed *UpdateP* is then at most $12\Delta(\sigma + 6\mu) + 18\mu + 1$.

7 COROLLARY 2. *The algorithm* PolyMatch *converges in* $O(n^2)$ *moves under the adversarial distributed daemon and in a general graph, provided that an underlying maximal matching has been initially built.*

The Manne *et al.* algorithm [MMPT09] builds a self-stabilizing maximal matching under the adversarial distributed daemon in a general graph, in $O(m)$ moves. This leads to a $O(m.n^2)$ moves complexity to build a 1-maximal matching with our algorithm without any assumption of an underlying maximal matching.

Now, the next section is devoted to the description of the technical proof.

5.7.1 *A matched node can write True in its end-variable at most twice*

The first three lemmas are technical lemmas.

LEMMA 14. *Let u be a matched node. Consider an execution $\mathcal{E}$ starting after u executed some rule. Let C be any configuration in $\mathcal{E}$. If $end_u = True$ in C then $s_u = True$ as well.*

Proof of 14. Let $C_0 \mapsto C_1$ be the transition in $\mathcal{E}$ in which u executed a rule for the last time before C. Observe that C may be equal to C_1. The executed rule is necessarily a *match* rule, otherwise end_u could not be $True$ in C_1. If it is a *MatchSecond* the lemma holds since in that case s_u is a copy of end_u. Assume now it is a *MatchFirst*. For end_u to be $True$ in C_1, $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_{m_u} = AskSecond(m_u)$ must hold in C_0, according to the guard of *MatchFirst*. This implies that u writes $True$ in s_u in transition $C_0 \mapsto C_1$. 14

LEMMA 15. *Let u be a matched node. Consider an execution $\mathcal{E}$ starting after u executed some rule. Let C be any configuration in $\mathcal{E}$. In C, if $s_u = True$ then $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$.*

Proof of 15. Consider transition $C_0 \mapsto C_1$ in which u executed a rule for the last time before C. The executed rule is necessarily a *match* rule, otherwise s_u could not be $True$ in C_1. Observe now that whichever *match* rule is applied, $Ask(u) \neq null$ – let us assume $Ask(u) = x$ – and $p_u = x$ and $p_x = u$ must hold in C_0 for s_u to be $True$ in C_1. $p_u = x$ still holds in C_1 and until C. Moreover, x must be in $single(N(u))$, otherwise u would have executed an *Update* instead of a match rule in $C_0 \mapsto C_1$, since *Update* has the higest priority among all rules. Finally, in transition $C_0 \mapsto C_1$, x cannot execute $UpdateP$ nor *ResetEnd* since $p_x \in matched(N(x)) \wedge p_{p_x} = x$ holds in C_0. Thus in C_1, $p_u = x$ and $p_x = u$ holds. Using the same argument, x cannot execute $UpdateP$ nor *ResetEnd* between configurations C_1 and C. Thus $p_u = x \wedge p_x = u$ in C. 15

LEMMA 16. *Let u be a matched node and $\mathcal{E}$ be an execution containing a transition $C_0 \mapsto C_1$ where u makes a move. From C_1, the predicate in the first line of the guard of the Update rule will ever hold from C_1.*

Proof of 16. Let C_2 be any configuration in $\mathcal{E}$ such that $C_2 \geqslant C_1$. Let $C_{10} \mapsto C_{11}$ be the last transition before C_2 in which u executes a move. Notice that by definition of $\mathcal{E}$, this transition exists. Assume by contradiction that one of the following predicates holds in C_2.

a) $(\alpha_u > \beta_u) \vee (\alpha_u, \beta_u \notin (single(N(u)) \cup \{null\})) \vee (\alpha_u = \beta_u \wedge \alpha_u \neq null)$
b) $p_u \notin (single(N(u)) \cup \{null\})$

By definition between C_{11} and C_2, u does not execute rules. To modify the variables α_u, β_u and p_u, u must execute a rule. Thus one of the two predicates also holds in C_{11}.

We first show that if predicate (1) holds in C_{11} then we get a contradiction. If u executes an $Update$ rule in transition $C_{10} \mapsto C_{11}$, then by definition of the *BestRematch* function, predicate (1) cannot hold in C_{11} (observe that the only way for $\alpha_u = \beta_u$ is when $\alpha_u = \beta_u = null$). Thus assume that u executes a *match* or *ResetMatch* rule. Notice that these rules do not modify the value of the α_u and β_u variables. This implies that if u executes one of these rules in $C_{10} \mapsto C_{11}$, predicate (1) not only hold in C_{11} but also in C_{10}. Observe that this implies, in that case that u is eligible for $Update$ in $C_{10} \mapsto C_{11}$, which gives the contradiction since $Update$ is the rule with the highest priority among all rules.

Now assume predicate (2) holds in C_{11}. In transition $C_{10} \mapsto C_{11}$, u cannot execute $Update$ nor $ResetMatch$ as this would imply that $p_u = null$ in C_{11}. Assume that in $C_{10} \mapsto C_{11}$ u executes a *match* rule. Since in C_{11}, $p_u \notin (single(N(u)) \cup \{null\})$ this implies that in C_{10}, $Ask(u) \notin (single(N(u)) \cup \{null\})$. This implies that $\alpha_u, \beta_u \notin (single(N(u)) \cup \{null\})$ in C_{10}. Thus u is eligible for $Update$ in transition $C_{10} \mapsto C_{11}$ and this yields the contradiction since $Update$ is the rule with the highest priority among all rules.

Since these two predicates cannot hold in C_2, this concludes the proof. 16

Now, we focus on particular configurations for a matched edge (u, v) corresponding to the fact they have completely exploited a 3-augmenting path.

LEMMA 17. *Let (u, v) be a matched edge, $\mathcal{E}$ be an execution and C be a configuration of $\mathcal{E}$. If in C, we have:*

a) $p_u \in single(N(u)) \wedge p_u = AskFirst(u) \wedge p_{p_u} = u$;
b) $p_v \in single(N(v)) \wedge p_v = AskSecond(v) \wedge p_{p_v} = v$;
c) $s_u = end_u = s_v = end_v = True$;

then neither u nor v will ever be eligible for any rule from C.

Proof of 17. Observe first that neither u nor v are eligible for any rule in C. Moreover, p_u (resp. p_v) is not eligible for an $UpdateP$ move since u (resp. v) does not make any move. Thus p_{p_u} and p_{p_v} will remain constant since u and v do not make any move and so neither u nor v will ever be eligible for any rule from C. 17

The configuration C described in Lemma 17 is called a $stop_{uv}$ configuration. From such a configuration neither u nor v will ever be eligible for any rule.

In Lemmas 19 and 20, we consider executions where a matched node u writes $True$ in end_u twice, and we focus on the transition $C_0 \mapsto C_1$ where u performs its second writing. Lemma 19 shows that, if u is First in C_0, then C_1 is a $stop_{um_u}$ configuration. Lemma 20 shows that, if u is Second in C_0, then either C_1 is a $stop_{um_u}$ configuration or it exists a configuration C_3 such that $C_3 > C_1$, u does not make any move from C_1 to C_3 and C_3 is a $stop_{um_u}$ configuration.

Lemma 18 and Corollary 4 are required to prove Lemmas 19 and 20.

LEMMA 18. *Let (u, v) be a matched edge. Let $\mathcal{E}$ be some execution in which v does not execute any rule. If it exists a transition $C_0 \mapsto C_1$ in $\mathcal{E}$ where u writes $True$ in end_u, then u is not eligible for any rule from C_1.*

Proof of 18. To write $True$ in end_u in transition $C_0 \mapsto C_1$, u must have executed a *match* rule. According to this rule, $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds C_0 with $p_u \in single(N(u))$, otherwise u would have executed an *Update* instead of a *match* rule. Now, in $C_0 \mapsto C_1$, p_u cannot execute *UpdateP* then it cannot change its p-value and v does not execute any move then it cannot change $Ask(u)$. Thus, $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds in both C_0 and C_1.

Assume now by contradiction that u executes a rule after configuration C_1. Let $C_2 \mapsto C_3$ be the next transition in which it executes a rule. Recall that between configurations C_1 and C_2 both u and v do not execute rules. Observe also that p_u is not eligible for $UpdateP$ between these configurations. Thus $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds from C_0 to C_2. Moreover the following points hold as well between C_0 and C_2 since in $C_0 \mapsto C_1$ u executed a *match* rule and v does not apply rules in $\mathcal{E}$:

- α_u, α_v, β_u and β_v do not change.
- The values of the variables of v do not change.
- $Ask(u)$ and $Ask(v)$ do not change.
- If u was *First* in C_0 it is *First* in C_2 and the same holds if it was *Second*.

Using these remarks, we start by proving that u is not eligible for *ResetMatch* in C_2. If it is *First* in C_2, this holds since $AskFirst(u) \neq null$ and $AskSecond(u) = null$. If it is *Second* then to be eligible for *ResetMatch*, $s_v = False$ must hold in C_2 since

$AskSecond(u) \neq null$. Since u executed $end_u = True$ in $C_0 \mapsto C_1$ and since u was *Second* in C_0, then necessarily $s_v = True$ in C_0 and thus in C_2 (using remark 2 above). So u is not eligible for *ResetMatch* in C_2.

We show now that u is not eligible for an *Update* in C_2. The α and β variables of u and v remain constant between C_0 and C_2. Thus if any of the three first disjunctions in the *Update* rule holds in C_2 then it also holds in C_0 and in $C_0 \mapsto C_1$ u should have executed an *Update* since it has higher priority than the *match* rules. Moreover since in C_2 ($p_u = Ask(u) \wedge p_{p_u} = u$) holds, the last two disjunctions of *Update* are *False* and we can state u is not eligible for this rule.

We conclude the proof by showing that u is not eligible for a *match* rule in C_2. If u was *First* in C_0 then it is *First* in C_2. To write *True* in end_u then ($p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = AskSecond(m_u) \wedge end_{m_u}$) must hold in C_0. Since in $C_0 \mapsto C_1$ v does not execute rules, it also holds in C_1. The same remark between configurations C_1 and C_2 implies that this predicate holds in C_2. Thus in C_2, all the three conditions of the *MatchFirst* guard are *False* and u not eligible for *MatchFirst*. A similar remark if u is *Second* implies that u will not be eligible for *MatchSecond* in C_2 if it was *Second* in C_0. 18

COROLLARY 4. *Let (u, v) be a matched edge. In any execution, if u writes True in end_u twice, then v executes a rule between these two writing.*

LEMMA 19. *Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes True in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes True in end_u for the second time in $\mathcal{E}$. If u is First in C_0 then the following holds:*

a) *in configuration C_0,*
   i) $s_v = end_v = True$;
   ii) $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u = True \wedge p_v = AskSecond(v)$;
   iii) $p_u \in single(N(u))$;
   iv) $p_v \in single(N(v)) \wedge p_{p_v} = v$;

b) *v does not execute any move in $C_0 \mapsto C_1$;*

c) *in configuration C_1,*
   i) $s_u = end_u = True$;
   ii) $p_u \in single(N(u)) \wedge p_v \in single(N(v))$;
   iii) $s_v = end_v = True$;
   iv) $p_u = AskFirst(u) \wedge p_v = AskSecond(v)$;
   v) $p_{p_u} = u \wedge p_{p_v} = v$.

Proof of 19. We prove Point 1a. Observe that for u to write *True* in end_u, end_v must be *True* in C_0. By Lemma 14 this implies that s_v is True as well. Now Point 1b holds by definition of the *MatchFirst* rule. As in C_0, u already executed an action, then according to Lemma 16, Point 1c holds and will always hold. By Corollary 4, u cannot write *True* consecutively if v does not execute moves. Thus at some point before C_0, v applied some rule. This implies that in configuration C_0, since $s_v = True$, by Lemma 15, $\exists x \in single(N(v)) : p_v = x \wedge p_x = v$. Thus Point 1d holds.

We now show that v does not execute any move in $C_0 \mapsto C_1$ (Point 2). Recall that v already executed an action before C_0, so by Lemma 16, line 1 of the $Update$ guard does not hold in C_0. Moreover, by Point 1d, line 2 does not hold either. Thus, v is not eligible for $Update$ in C_0. We also have that $s_u = True$ and $AskSecond(v) \neq null$ in C_0, thus v is not eligible for *ResetMatch*. Observe now that by Points 1a, 1b and 1d, v is not eligible for *MatchSecond* in C_0. Finally v cannot execute *MatchFirst* since $AskFirst(v) = null$. Thus v does not execute any move in $C_0 \mapsto C_1$ and so Point 2 holds.

In C_1, end_u is True by hypothesis and according to Point 1b, u writes $True$ in s_u in transition $C_0 \mapsto C_1$. Thus Point 3a holds. Points 3b holds by Points 1c and 1d. Points 3c holds by Points 1a and 2. $AskFirst(u)$ and $AskSecond(v)$ remain constant in $C_0 \mapsto C_1$ since neither u nor v executes an $Update$ in this transition. Moreover p_v remains constant in $C_0 \mapsto C_1$ by Point 2 and p_u remains constant also since it writes $AskFirst(u)$ in p_u in this transition while $p_u = AskFirst(u)$ in C_0. Thus Points 3d holds. Observe that nor p_u neither p_v is eligible for an $UpdateP$ in C_0, thus Point 3e holds. 19

Now, we consider the case where u is Second.

LEMMA 20. *Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes $True$ in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes $True$ in end_u for the second time in $\mathcal{E}$. If u is Second in C_0 then the following holds:*

a) *in configuration C_0,*
   i) $s_v = True \wedge p_v = AskFirst(v)$;
   ii) $p_v \in single(N(v)) \wedge p_{p_v} = v$;
b) *in transition $C_0 \mapsto C_1$, v is not eligible for $Update$ nor $ResetMatch$;*
c) *in configuration C_1,*
   i) $s_u = end_u = True$;
   ii) $p_v \in single(N(v)) \wedge p_v = AskFirst(v) \wedge p_{p_v} = v$;
   iii) $p_u \in single(N(u)) \wedge p_u = AskSecond(u) \wedge p_{p_u} = u$;
   iv) $s_v = True$;
d) *u is not eligible for any move in C_1;*
e) *If $end_u = False$ in C_1 then the following holds:*
   i) *From C_1, v executes a next move and this move is a $MatchFirst$;*
   ii) *Let us assume this move (the first move of v from C_1) is done in transition $C_2 \mapsto C_3$. In configuration C_3, we have:*
      A) $s_u = end_u = True$;
      B) $p_v \in single(N(v)) \wedge p_v = AskFirst(v) \wedge p_{p_v} = v$;
      C) $p_u \in single(N(u)) \wedge p_u = AskSecond(u) \wedge p_{p_u} = u$;
      D) $s_v = True$;
      E) *u does not execute moves between C_1 and C_3;*
      F) $end_v = True$;

Proof of 20. We show Point 1a. For u to write $True$ in transition $C_0 \mapsto C_1$, u executes a *MatchSecond* in this transition. Thus $s_v = True$ must hold in C_0 and $p_v = AskFirst(v)$ as well. By Corollary 4, u cannot write $True$ consecutively if v does not execute any move. Thus at some point before C_0, v applied some rule. Thus, and by Lemma 15, $\exists x \in single(N(v)) : p_v = x \wedge p_x = v$ in configuration C_0, so Point 1b holds.

As $AskFirst(v) \neq null$ in C_0, v is not eligible for *ResetMatch* in C_0. We prove now that v is not eligible for $Update$. By Corollary 4 and Lemma 16, line 1 of the $Update$ guard does not hold in C_0. Finally, according to Point 2b, the second line of the $Update$ guard does not hold, which concludes Point 2.

We consider now Point 3a. In C_1, $s_u = end_u = True$ holds because, executing a *MatchSecond*, u writes $True$ in end_u and writes end_u in s_u during transition $C_0 \mapsto C_1$.

We now show Point 3b. $AskFirst(v)$ and $AskSecond(u)$ remain constant in $C_0 \mapsto C_1$ since neither u nor v execute an $Update$ in this transition. Moreover, the only rule v can execute in $C_0 \mapsto C_1$ is a *MatchFirst*, according to Point 2. Thus v does not change its p-value in $C_0 \mapsto C_1$ and so $p_v = AskFirst(v)$ in C_1. Now, in C_0, $v \in matched(N(p_v)) \wedge p_{p_v} = v$ thus p_v cannot execute $UpdateP$ in $C_0 \mapsto C_1$ and thus it cannot change its p-value. So, $p_{p_v} = v$ in C_1.

Point 3c holds since after u executed a *MatchSecond* in $C_0 \mapsto C_1$, observe that necessarily $p_u = AskSecond(u)$ in C_1. Moreover, $s_u = True$ in C_1 so, according to Lemma 15, $\exists y \in single(N(u)) : p_u = y \wedge p_y = u$ in C_1.

$p_v = AskFirst(v)$ and $p_{p_v} = v$ hold in C_0, according to Points 2a and 2b. Moreover, $p_u = AskSecond(u)$ holds in C_0 since u writes $True$ in end_u while executing a *MatchSecond* in $C_0 \mapsto C_1$. Finally, by Point 2, v can only execute *MatchFirst* in $C_0 \mapsto C_1$, thus variable s_v remains $True$ in transition $C_0 \mapsto C_1$ and Point 3d holds.

We now prove Point 4. If $end_v = True$ in C_1, then according to Lemma 17, u is not eligible for any rule in C_1. Now, let us consider the case $end_v = False$ in C_1. By Points 3c and 3d, u is not eligible for *ResetMatch*. By Point 3c and Lemma 16, u is not eligible for $Update$. By Points 3a, 3b and 3c, u is not eligible for *MatchSecond*. Finally, since u is Second in C_1, u is not eligible for *MatchFirst* neither and Point 4 holds.

Now since between C_1 and C_2, v does not execute any rule (by Point 5b), and since p_u (resp. p_v) is not eligible for $UpdateP$ while u (resp. v) does not move (because $p_{p_u} = u$ (resp. $p_{p_v} = v$)), then $Ask(u)$, $Ask(v)$, p_{p_u} and p_{p_v} remain constant while u does not make any move. And so, properties 3a, 3b, 3c and 3d hold for any configuration between C_1 and C_2, thus u is not eligible for any rule between C_1 and C_2 and u will not execute any move from C_1 to C_3. Moreover, the end_v-value is the same from C_1 to C_2.

If $end_v = False$ in C_2, then v is eligible for a *MatchFirst* and that it will write $True$ in its end_v-variable while all properties of Point 3 will still hold in C_3. Thus Point 5 holds. 20

THEOREM 11. *In any execution, a matched node u can write $end_u := True$ at most*

twice.

Proof of 11. Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes $True$ in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes $True$ in end_u for the second time in $\mathcal{E}$. If u is First (resp. Second) in C_0 then from Lemmas 17 and 19, (resp. 20), from C1, neither u nor v will ever be eligible for any rule. 11

5.7.2 *The number of times single nodes can change their end-variable*

In the following, μ denote the number of matched nodes and σ the number of single nodes.

LEMMA 21. *Let x be a single node. If x writes $True$ in some transition $C_0 \mapsto C_1$ then, in C_0, $\exists u \in matched(N(x)) : p_x = u \wedge p_u = x \wedge end_x = False \wedge end_u = True$.*

Proof of 21. To write True in its *end* variable, a single node must apply $UpdateEnd$. Observe now that to apply this rule, the conditions described in the Lemma must hold. 21

LEMMA 22. *Let u be a matched node. Consider an execution $\mathcal{E}$ starting after u executed some rule and in which end_u is always $True$, except for the last configuration D of $\mathcal{E}$ in which it may be False. Let $\mathcal{E}\backslash D$ be all configurations of $\mathcal{E}$ but configuration D. In $\mathcal{E}\backslash D$, the following holds:*

- $p_u \in single(N(u))$;
- p_u *remains constant.*

Proof of 22. Since $end_u = True$ in $\mathcal{E}\backslash D$, the last rule executed before $\mathcal{E}$ is necessarily a $Match$ rule. So, at the beginning of $\mathcal{E}$, $p_u \in single(N(u))$, otherwise, u would not have executed a $Match$ rule, but an $Update$ instead.

We prove now that in $\mathcal{E}\backslash D$, p_u remains constant. Assume by contradiction that there exists a transition in which p_u is modified. Let $C_0 \mapsto C_1$ be the first such transition. First, observe that in $\mathcal{E}\backslash D$, u cannot execute $ResetMatch$ nor $Update$ since that would set end_u to $False$. Thus u must execute a $Match$ rule in $C_0 \mapsto C_1$. Since the value of p_u changes in this transition, this implies that $Ask(u) \neq p_u$ in C_0. Thus, whatever the $Match$ rule, observe now that in C_1, end_u must be $False$, which gives a contradiction and concludes the proof. 22

DEFINITION 43. *Let u be a matched node. We say that a transition $C_0 \mapsto C_1$ is of type* "a single copies True from u" *if it exists a single node x such that $(p_x = u \wedge p_u = x \wedge end_x = False)$ in C_0 and $end_x = True$ in C_1. Notice that by Lemma 21, $end_u = True$ in C_0 and $x \in single(N(u))$.*

If a transition $C_0 \mapsto C_1$ is of type "a single node copies True from u" *and if x is the single node with $(p_x = u \wedge p_u = x \wedge end_x = False)$ in C_0 and $end_x = True$ in C_1, then we will say x* copies $True$ from u.

LEMMA 23. *Let u be a matched node and $\mathcal{E}$ be an execution. In $\mathcal{E}$, there are at most three transitions of type* "a single copies $True$ from u".

Proof of 23. Let $\mathcal{E}$ be an execution. We consider some sub-executions of $\mathcal{E}$.

Let $\mathcal{E}_{init}$ be a sub-execution of $\mathcal{E}$ that starts in the initial configuration of $\mathcal{E}$ and that ends just after the first move of u. Let $C_0 \mapsto C_1$ be the last transition of $\mathcal{E}_{init}$. Observe that u does not execute any move until configuration C_0 and executes its first move in transition $C_0 \mapsto C_1$. We will write $\mathcal{E}_{init} \setminus C_1$ to denote all configurations of $\mathcal{E}_{init}$ but the configuration C_1. We prove that there is at most one transition of type *"a single copies True from u"* in $\mathcal{E}_{init}$.

There are two possible cases regarding end_u in all configuration of $\mathcal{E}_{init} \setminus C_1$: either end_u is always $True$ or end_u is always $False$. If $end_u = False$ then by Definition 43, no single node can copy $True$ from u in $\mathcal{E}_{init}$, not even in transition $C_0 \mapsto C_1$, since no single node is eligible for such a copy in C_0. If $end_u = True$, once again, there are two cases: either (i) $(p_u = null \vee p_u \notin single(N(u)))$ in all configuration of $\mathcal{E}_{init} \setminus C_1$, or (ii) $(p_u \in single(N(u)))$ in $\mathcal{E}_{init} \setminus C_1$. In case (i) then by Definition 43 no single node can copy $True$ from u in $\mathcal{E}_{init}$, not even in $C_0 \mapsto C_1$. In case (ii), observe that p_u remains constant in all configurations of $\mathcal{E}_{init} \setminus C_1$, thus at most one single node can copy $True$ from u in $\mathcal{E}_{init}$.

Let $\mathcal{E}_{true}$ be a sub-execution of $\mathcal{E}$ starting after u executed some rule and such that: for all configurations in $\mathcal{E}_{true}$ but the last one, $end_u = True$. There is no constraint on the value of end_u in the last configuration of $\mathcal{E}_{true}$. According to Lemma 22, $p_u \in single(N(u))$ and p_u remains constant in all configurations of $\mathcal{E}_{true}$ but the last one. This implies that at most one single can copy $True$ from u in $\mathcal{E}_{true}$.

Let $\mathcal{E}_{false}$ be an execution starting after u executed some rule and such that: for all configurations in $\mathcal{E}_{false}$ but the last one, $end_u = False$. There is no constraint on the value of end_u in the last configuration of $\mathcal{E}_{false}$. By Definition 43, no single node will be able to copy $True$ from u in $\mathcal{E}_{false}$.

To conclude, by Corollary 11, u can write $True$ in its *end* variable at most twice. Thus, for all executions $\mathcal{E}$, $\mathcal{E}$ contains exactly one sub-execution of type $\mathcal{E}_{init}$, and at most two sub-executions of type $\mathcal{E}_{true}$ and the remaining sub-executions are of type $\mathcal{E}_{false}$. This implies that in total, we have at most three transitions of type *"a single copies True from u"* in $\mathcal{E}$. 23

LEMMA 24. *In any execution, the number of transitions where a single node writes True in its end variable is at most* 3μ.

Proof of 24. Let $\mathcal{E}$ be an execution and x be a single node. If x writes $True$ in end_x in some transition of $\mathcal{E}$, then x necessarily executes an $UpdateEnd$ rule and by Definition 43, this means *x copies True from some matched node* in this transition. Now the lemma holds by Lemma 23. 24

LEMMA 25. *In any execution, the number of transitions where a single node changes the value of its end variables (from True to False or from False to True) is at most* $\sigma + 6\mu$ *times.*

Proof of 25. A single node can write $True$ in its *end* variable at most 3μ times, by Corollary 24. Each of this writing allows one writing from $True$ to $False$, which leads

to 6μ possible modifications of the *end* variables. Now, let us consider a single node x. If $end_x = False$ initially, then no more change is possible, however if $end_x = True$ initially, then one more modification from $True$ to $False$ is possible. Each single node can do at most one modification due to this initialization and thus the Lemma holds. 25

5.7.3 *How many Update in an execution?*

DEFINITION 44. *Let u be a matched node and C be a configuration. We define $Cand(u, C) = \{x \in single(N(u)) : (p_x = u \vee end_x = False)\}$ which is the set of vertices considered by the function BestRematch(u) in configuration C.*

LEMMA 26. *Let u be a matched node that has already executed some rule. If there exists a transition $C_0 \mapsto C_1$ such that u is eligible for Update in C_1 and not in C_0, then there exists a single node x such that $x \in Cand(u, C_0) \backslash Cand(u, C_1)$ or $x \in Cand(u, C_1) \backslash Cand(u, C_0)$. Moreover, in transition $C_0 \mapsto C_1$, x flips the value of its end variable.*

Proof of 26. Since u has already executed some rule, to become eligible for *Update* in transition $C_0 \mapsto C_1$, necessarily the second disjonction in the *Update* rule must hold, by Lemma 16. This implies that $(\alpha_v, \beta_v) \neq BestRematch(v)$ must become $True$ in $C_0 \mapsto C_1$. Now either $Lowest(Cand(u, C_0)) \notin Cand(u, C_1)$ or $\exists x \notin Cand(u, C_0)$ such that $x = Lowest(Cand(u, C_1))$. This proves the first point.

For the second point we first consider the case $x \in Cand(u, C_1)$ and $x \notin Cand(u, C_0)$. Necessarily $end_x = True \wedge p_x \neq u$ in C_0 and $end_x = False \vee p_x = u$ in C_1. If $p_x = u$ in C_1 then in transition $C_0 \mapsto C_1$, x has executed an *UpdateP* and the second point holds. Assume now that $p_x \neq u$ in C_1. Necessarily $end_u = False$ in C_1 and the Lemma holds.

We consider the second case in which $x \notin Cand(u, C_1)$ and $x \in Cand(u, C_0)$. Necessarily in C_1, $p_x \neq v$ and $end_x = True$. Thus if $end_x = False$ in C_0 the lemma holds. Assume by contradiction that $end_x = True$ in C_0. This implies $p_x = u$ in C_0. But since in C_1 $p_x \neq u$ then x executed either *UpdateP* or *UpdateEnd* in $C_0 \mapsto C_1$ which implies $end_x = False$ in C_1, a contradiction. This completes the proof. 26

COROLLARY 5. *Matched nodes can execute at most $\Delta(\sigma + 6\mu) + \mu$ times the Update rule.*

Proof of 5. Initially each matched node can be eligible for an *Update*. Now, let us consider only matched nodes that have already executed a move. For such a node to become eligible for an *Update* rule, at least one single node must change the value of its *end* variable by Lemma 26. Thus, each change of the *end* value of a single node can generate at most Δ matched nodes to be eligible for an *Update*. By Lemma 25, the number of transitions where a single node changes the value of its *end* variables is at most $\sigma + 6\mu$ times. Thus we obtain at most $\Delta(\sigma + 6\mu)$ *Update* generated by a change of the *end* value of a single node and the Lemma holds. 5

DEFINITION 45. *Let (u, v) be a matched edge. In the following, we call $\mathcal{F}$, a finite execution where neither u nor v execute the Update rule. Let $D_\mathcal{E}$ be the first configuration of $\mathcal{F}$ and $D'_\mathcal{E}$ be the last one.*

Observe that in the execution $\mathcal{F}$, all variables α and β of nodes u and v remain constant and thus, predicates *AskFirst* and *AskSecond* for these two nodes remain constant too.

LEMMA 27. *If $Ask(u) = Ask(v) = null$ in $\mathcal{F}$, then u and v can both execute at most one ResetMatch.*

Proof of 27. Recall that in the execution $\mathcal{F}$, by definition, u and v do not execute the *Update* rule. Moreover, these two nodes are not eligible for *Match* rules since $Ask(u) = Ask(v) = null$. Thus they are only eligible for *ResetMatch*. Observe now it is not possible to execute tis rule twice in a row, which completes the proof. 27

LEMMA 28. *Assume that in $\mathcal{F}$, u is First and v is Second. If s_u is False in all configurations of $\mathcal{F}$ but the last one, then v can execute at most one rule in $\mathcal{F}$.*

Proof of 28. Since $s_u = False$ in all configurations of $\mathcal{F}$ but the last one, node v which is *Second* can only be eligible for *ResetMatch*. Observe that if v executes *ResetMatch*, it is not eligible for a rule anymore and the Lemma holds. 28

LEMMA 29. *Assume that in $\mathcal{F}$, u is First and v is Second. If s_u is False throughout $\mathcal{F}$, then u can execute at most one rule in $\mathcal{F}$.*

Proof of 29. Node u can only be eligible for *MatchFirst*. Assume u executes *MatchFirst* for the first time in some transition $C_0 \mapsto C_1$, then in C_1, necessarily, $p_u = AskFirst(u)$, $s_u = False$ (by hypothesis) and $end_u = False$ by Lemma 14. Let $\mathcal{F}_1$ be the execution starting in C_1 and finishing in $D'_\mathcal{E}$. Since in $\mathcal{F}_1$, there is no *Update* of nodes u and v, observe that $p_u = AskFirst(u)$ remains $True$ in this execution. Assume by contradiction that u executes another *MatchFirst* in $\mathcal{F}_1$. Consider the first transition $C_2 \mapsto C_3$ after C_1 when it executes this rule. Notice that between C_1 and C_2 it does not execute rules. Thus in C_2, $p_u = AskFirst(u)$, $s_u = False$ and $end_u = False$ hold. Now if u executes *MatchFirst* in C_2 it is necessarily to modify the value of s_u or end_u. By definition, it cannot change the value of s_u. Moreover it cannot modify the value of end_u as this would imply by Lemma 14 that $s_u = True$ in C_3. This completes the proof. 29

LEMMA 30. *Let (u, v) be a matched edge. Assume that in $\mathcal{F}$, u is First, v is Second and that u writes True in s_u in some transition of $\mathcal{F}$. Let $C_0 \mapsto C_1$ be the transition in $\mathcal{F}$ in which u writes True in s_u for the first time. Let $\mathcal{F}_1$ be the execution starting in C_1 and finishing in $D'_\mathcal{E}$. In $\mathcal{F}_1$, u can apply at most 3 rules and v at most 2.*

Proof of 30. We first prove that in $\mathcal{T}_1, s_u$ remains $True$. Observe that u cannot execute $Update$ neither $ResetMatch$ since it is $First$. So u can only execute $MatchFirst$ in $\mathcal{T}_1$. For u to write $False$ in s_u, it must exists a configuration in $\mathcal{T}_1$ such that $p_u \neq AskFirst(u) \vee p_{p_u} \neq u \vee p_v \notin \{AskSecond(v), null\}$. Let us prove that none of these cases are possible.

Since u executed $MatchFirst$ in transition $C_0 \mapsto C_1$ writting $True$ in s_u then, by definition of this rule, $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_v \in \{AskSecond(v), null\}$ holds in C_0. As there is no $Update$ of u and v in $\mathcal{T}$, then $AskFirst(u)$ and $AskSecond(v)$ remain constant throughout $\mathcal{T}$ (and $\mathcal{T}_1$). So each time u executes a $MatchFirst$, it writes the same value $AskFirst(u)$ in its p-variable. Thus $p_u = AskFirst(u)$ holds throughout $\mathcal{T}_1$. Moreover, each time v executes a rule, it writes either $null$ or the same value $AskSecond(v)$ in its p-variable. Thus $p_v \in \{AskSecond(v), null\}$ holds throughout $\mathcal{T}_1$. Now by Lemma 15, in C_1 we have, $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$, since $s_u = True$. This stays $True$ in $\mathcal{T}_1$ as p_u remains constant and x will then not be eligible for $UpdateP$ in $\mathcal{T}_1$. Thus $p_{p_u} = u$ holds throughout $\mathcal{T}_1$. Thus, $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_v \in \{AskSecond(v), null\}$ holds throughout $\mathcal{T}_1$ and so $s_u = True$ throughout $\mathcal{T}_1$.

This implies that in $\mathcal{T}_1$, v is only eligible for $MatchSecond$. The first time it executes this rule in some transition $B_0 \mapsto B_1$, with $B_1 \geqslant C_1$, then in B_1, $p_v = AskSecond(v)$, $s_v = end_v$ and this will hold between B_1 and $D'_\mathcal{E}$. If $end_v = True$ in B_1 then this will stay $True$ between B_1 and $D'_\mathcal{E}$. Indeed, p_v is not eligible for $UpdateP$ and we already showed that $p_u = AskFirst(u)$ holds in $\mathcal{T}_1$. In that case, between B_1 and $D'_\mathcal{E}$, v will not be eligible for any rule and so v will have executed at most one rule in $\mathcal{T}_1$. In the other case, that is $end_v (= s_v) = False$ in B_1, since $p_v = AskSecond(v)$ holds between B_1 and $D'_\mathcal{E}$, necessarily, the next time v executes a $MatchSecond$ rule, it is to write $True$ in end_v. After that observe that v is not eligible for any rule. Thus, v can execute at most 2 rules in $\mathcal{T}_1$.

To conclude the proof it remains to count the number of moves of u in $\mathcal{T}_1$. Recall that we proved s_u is always $True$ in $\mathcal{T}_1$. Thus whenever u executes a $MatchFirst$, it is to modify the value of its end variable. Observe that this value depends in fact of the value of end_v and of p_v since we proved $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_v \in \{AskSecond(v), null\}$ holds throughout $\mathcal{T}_1$. Since we proved that in $\mathcal{T}_1$, v can execute at most two rules, this implies that these variables can have at most three different values in $\mathcal{T}_1$. Thus u can execute at most 3 rules in $\mathcal{T}_1$. 30

LEMMA 31. *Assume that in $\mathcal{T}$, u is First and v is Second. If s_u is True throughout $\mathcal{T}$ and if u does not execute any move in $\mathcal{T}$, then v can execute at most two rules in $\mathcal{T}$.*

Proof of 31. By Definition 45, v cannot execute $Update$ in $\mathcal{T}_1$. Since we suppose that in $\mathcal{T}_1$, $s_u = True$ then v is not eligible for $ResetMatch$. Thus in $\mathcal{T}_1$, v can only execute $MatchSecond$. After it executed this rule for the first time, $p_v = AskSecond(v)$ and $s_v = end_v$ will always hold, since v is only eligible for $MatchSecond$. Thus the second time it executes this rule, it is necessarily to modify its end_v and s_v variables. Observe that after that, since u does not execute rules, v is not eligible for any rule. 31

LEMMA 32. *In $\mathcal{T}$, u and v can globally execute at most* 12 *rules.*

Proof of 32. If $Ask(u) = Ask(v) = null$, the Lemma holds by Lemma 27. Assume now that u is $First$ and v $Second$. We consider two executions in $\mathcal{T}$.

Let $C_0 \mapsto C_1$ be the first transition in $\mathcal{T}$ in which u executes a rule. Let $\mathcal{T}_0$ be the execution starting in $D_\mathcal{E}$ and finishing in C_0. There are two cases.

If $s_u = False$ in $\mathcal{T}_0$ then v is only eligible for $ResetMatch$ in this execution. Observe that after it executes this rule for the first time in $\mathcal{T}_0$, it is not eligible for any rule after that in $\mathcal{T}_0$.

If $s_u = True$ in $\mathcal{T}_0$ then by Lemma 31, v can execute at most two rules in this execution. In transition $C_0 \mapsto C_1$, u and v can execute one rule each.

Let $\mathcal{T}_1$ be the execution starting in C_1 and finishing in $D'_\mathcal{E}$. Whatever rule u executes in transition $C_0 \mapsto C_1$ observe that u either writes $True$ or $False$ in s_u. If u writes $True$ in s_u in transition $C_0 \mapsto C_1$, then by Lemma 30, u and v can execute at most five rules in total in $\mathcal{T}_1$.

Consider the other case in which u writes $False$ in C_1. Let $C_2 \mapsto C_3$ be the first transition in $\mathcal{T}_1$ in which u writes $True$ in s_u. Call $\mathcal{T}_{10}$ the execution between C_1 and C_3 and $\mathcal{T}_{11}$ the execution between C_3 and $D'_\mathcal{E}$. By definition, s_u stays $False$ in $\mathcal{T}_{10} \backslash C_3$. Thus in $\mathcal{T}_{10} \backslash C_3$, u can execute at most one rule, by Lemma 29. Now in $\mathcal{T}_{10}$, u can execute at most two rules. By Lemma 28, v can execute at most one rule in $\mathcal{T}_{10}$. In total, u and v can execute at most three rules in $\mathcal{T}_{10}$. In $\mathcal{T}_{11}$, u and v can execute at most five rules by Lemma 30. Thus in $\mathcal{T}_1$, u and v can apply at most eight rules. 32

THEOREM 12. *In any execution, matched nodes can execute at most* $12\Delta(\sigma+6\mu)+18\mu$ *rules.*

Proof of 12. Let k be the number of edges in the underlying maximal matching, $k = \frac{\mu}{2}$. For $i \in [1, .., k]$, let $\{(u_i, v_i) = a_i\}$ be the set of matched edges. By $Update(a_i)$ we denote an $Update$ rule executed by node u_i or v_i. By Lemma 32, between two $Update(a_i)$ rules, nodes u_i and v_i can execute at most 12 rules. By Corollary 5, there are at most $\Delta(\sigma + 6\mu) + \mu$ executed $Update$ rules. Thus in total, nodes can execute at most $\sum_{i=1}^{k} 12 \times (\#Update(a_i) + 1)$

$= 12 \sum_{i=1}^{k} \#Update(a_i) + 12 \sum_{i=1}^{k} 1 \leqslant 12(\Delta(\sigma + 6\mu) + \mu) + 12k = 12\Delta(\sigma + 6\mu) + 18\mu$ rules. 12

LEMMA 33. *In any execution, single nodes can execute at most σ times the ResetEnd rule.*

Proof of 33. We prove that a single node x can execute the $ResetEnd$ rule at most once. Assume by contradiction that it executes this rule twice. Let $C_0 \mapsto C_1$ be the transition when it executes it the second time. In C_0, $end_x = True$, by definition of the rule. Since x already executed a $ResetEnd$ rule, it must have some point wrote $True$ in end_x. This is only possible through an execution of $UpdateEnd$. Thus consider the last transition $D_0 \mapsto D_1$ in which it executed this rule. Observe that $D_1 \leqslant C_0$.

Since between D_1 and C_0, end_x remains $True$, observe that x does not execute any rule between these two configurations. Now since in D_1, $p_x \neq null$ and this holds in C_0 then x is not eligible for *ResetEnd* in C_0, which gives the contradiction. This implies that single nodes can execute at most $O(\sigma)$ times the *ResetEnd* rule. 33

LEMMA 34. *In any execution, single nodes can execute at most $\sigma + 6\mu$ times the UpdateEnd rule.*

Proof of 34. By Lemma 25, single nodes can change the value of their *end* variable at most $\sigma + 6\mu$ times. Thus they can apply *UpdateEnd* at most $\sigma + 6\mu$ times, since in every application of this rule, the value of the *end* variable must change. 34

LEMMA 35. *In any execution, single nodes can execute at most $12\Delta(\sigma + 6\mu) + 18\mu + 1$ times the UpdateP rule.*

Proof of 35. Let x be a single node. Let $C_0 \mapsto C_1$ be a transition in which x executes an *UpdateP* rule and let $C_2 \mapsto C_3$ be the next transition after C_1 in which x executes an *UpdateP* rule. We prove that for x to execute the *UpdateP* rule in $C_2 \mapsto C_3$, a matched node had to execute a move between C_0 and C_2.

In C_1 there are two cases: either $p_x = null$ or $p_x \neq null$. Assume to begin that $p_x = null$. This implies that in C_0 the set $\{w \in N(x) | p_w = x\}$ is empty. In C_2, $p_x = null$, since between C_1 and C_2, x can only apply *UpdateEnd* or *ResetEnd*. Thus if it applies *UpdateP* in C_2, necessarily $\{w \in N(x) | p_w = x\} \neq \emptyset$. This implies that a matched node must have executed a *Match* rule between C_1 and C_2 and the lemma holds in that case.

Consider now the case in which $p_x = u$ with $u \neq null$ in C_1. By definition of the *UpdateP* rule, we also have $u \in matched(N(x)) \wedge p_u = x$ holds in C_0. In C_2 we still have that $p_x = u$ since between C_1 and C_2, x can only execute *UpdateEnd* or *ResetEnd*. Thus if x executes *UpdateP* in C_2, necessarily $p_{p_x} \neq x$. This implies that $p_u \neq x$ and so u executed a rule between C_0 and C_2.

Now, the lemma holds by Theorem 12. 35

COROLLARY 6. *In any execution, nodes can execute at most $O(n^2)$ moves.*

Proof of 6. According to Lemmas 33, 34 and 35, single nodes can execute at most $O(n^2)$ moves. Moreover, according to Theorem 12, matched nodes can execute at most $O(n^2)$ moves. 6

COROLLARY 7. *The algorithm PolyMatch converges in $O(n^2)$ moves under the adversarial distributed daemon and in a general graph, provided that an underlying maximal matching has been initially built.*

Recall that the algorithm PolyMatch assumes an underlying maximal matching. As we said in section 5.2, we can use the self-stabilizing maximal matching algorithm from Manne *et al.* [MMPT09] that stabilizes in $O(m)$ moves. Then, using a classical composition of these two algorithms [Dolo0], we obtain a total time complexity in $O(m \times n^2)$ moves under the adversarial distributed daemon.

5.8 CONCLUSION

In this chapter we have presented a proof that the Manne et al. algorithm reaches a sub-exponential bound in terms of moves. This bound was, until now, supposed to be tight. From here it was natural to ask whether it is possible to do better by bringing the running time to polynomial. This was done successfully by exhibiting a new self-stabilizing algorithm that reaches a 2/3−approximation of the maximum matching in $O(n^2)$ moves, under the distributed adversarial daemon without particular topology assumptions. The new algorithm adapts and extends on the algorithm by Manne et al. by using a new variable *end*. Using this variable we can distinguish between worth destroying 3−augmented paths and the others, by making a node signal to its neighbours when it is done exploiting the path it detected. This signalling prevents neighbouring nodes from doing the immediate and systematic resetting that led to the destruction of possibly valid 3−augmenting paths in the Manne et al. algorithm. It is this avoidance that spares the new algorithm the necessary moves to reach its polynomial convergence time.

We can ask whether this same scheme (using the *end*) variable, can be used to design new algorithms that use the general version of Karp's theorem. That is by detecting t−augmenting paths for $t > 3$ to reach a better approximation. This doesn't seem trivial, as the usage of the *end* variable implies having to synchronize all the nodes across the augmenting path. Therefore the longer the detected augmenting path, the longer the number of nodes to synchronize in the graph.

Self-stabilizing publish/Subscribe systems 6

This chapter focuses on the more practical problem of routing messages in a publish/subscribe system on wireless sensor networks. In this case, the entities of the distributed system have limited computing power and a highly dynamic neighbourhood. We give a self-stabilizing algorithm for this task and the result of its implementation in a simulated setting. This work appeared in [STM15]

6.1 INTRODUCTION

In a wireless sensor network (WSN) with nodes sensing the environment and actuators reacting to these measurements, the need for a well structured data dissemination architecture arises. A naive approach is to flood the entire network with that data. Such an approach has the advantage of not requiring an infrastructure. Nevertheless, this approach clearly does not scale with the number of nodes and frequency of information events.

An effective solution for this task is provided by publish/subscribe systems [EFGK03]. Formally, the publish/subscribe paradigm describes a loosely coupled distributed information dissemination middleware. In this paradigm, senders (i.e., publishers) of data do not directly assign recipients (i.e., subscriber) to a message. Instead, publications are routed asynchronously by the system's infrastructure to all nodes which registered their interest. Interests refer either to the message content filtered by subscribers, or by a categorization done by the publisher. Such categories are called channels or topics.

The publish/subscribe paradigm exists for quite some time and has been widely studied for the Internet environment, e.g., [CDKR02, CRW01]. This paradigm in WSNs, however, is fundamentally different and much more challenging. The reason is that WSNs are resource-constrained and operate based on multi-hop relay and ad hoc routing rather than on a robust infrastructure as the Internet. Publish/subscribe systems must address scalability in terms of the number of subscriptions, messages, and storage footprint. Furthermore, fault tolerance is an important design consideration. Faults can hit the routing infrastructure leading to lost messages/subscriptions or to the delivery of unwanted messages. These two aspects are of particular importance for WSNs, since it is well known that links are error-prone, nodes are unreliable, and that the available memory resources are limited.

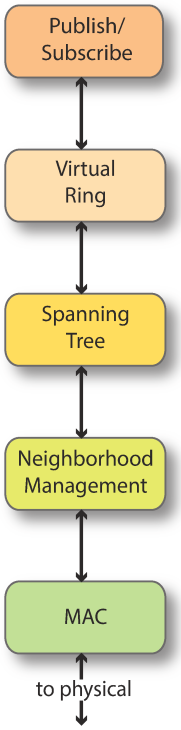


FIGURE 6.1
Layered architecture

The main contribution is a scalable, self-stabilizing middleware for channel-based publish/subscribe applications in WSNs. According to the self-stabilizing paradigm starting from an arbitrary state, the middleware eventually provides safety and liveness properties such as the guaranteed delivery of all published messages to all subscribers of the corresponding channel and the correct handling of subscriptions and unsubscriptions. We consider different kinds of faults, apart from message and memory corruptions, we also respect network changes such as node and link removals and additions.

The design of the middleware is based on a layered architecture (see Fig. 6.1), each layer is self-stabilizing with respect to its service which is exposed by an API. The API of the top layer provides functions to (un)subscribe to a channel and to publish messages to it. This functionality is enabled by the virtual ring layer. This is a virtual network structure which arranges all nodes on a ring. The publish/subscribe system uses this structure to route messages. We augment this architecture by introducing *short-cuts*, which enable the avoidance of sections of the ring without subscribers for a particular channel to improve scalability. Formation and maintenance of the virtual ring are based on a tree, which is realized using a well-known self-stabilizing spanning tree algorithm. The basis of our architecture is provided by the neighbourhood protocol Mahalle$^+$ [STW$^+$13] which is self-stabilizing as well. It constructs a stable topology while acting agile. Agility ensures that the protocol adjusts quickly to new or failing nodes and links. On the other hand, transient faults, like burst errors, are

ignored to make the topology stable.

An important feature of Mahalle$^+$ is that it allows the specification of an upper bound C_N for the number of neighbours of each node. Therefore, C_N generates a trade-off between the average route length of publications and the memory space required for routing-tables. With C_N, the former grows linearly, while the later grows quadratically. Furthermore, increasing C_N introduces a higher number of short-cuts, hence, reduces path lengths.

Self-stabilization is achieved using, besides classical techniques, the concept of *leasing*, in particular for routing entries and subscriptions [GC89]. Before a leasing period expires a routing entry has to be renewed, or it will be discarded and therefore removed from any corresponding table. Changes in lower layers trigger update operation in dependent layers. This work is an extension of [SMT14].

6.2 RELATED WORK

Channel-based publish/subscribe systems are often realized as an application-level overlay of brokers connected in a peer-to-peer manner. The overlay infrastructure directly impacts performance and scalability, such as the message routing cost. Chockler et al. try to minimize the complexity of the overlay by organizing all nodes interested in the same channel into a tree with low diameter [CMTV07]. They introduce a new optimization problem, called Minimum Topic-Connected Overlay capturing the trade-off between the scalability of the overlay and the message forwarding overhead as follows: For a set of subscriptions to different channels, connect the nodes using the minimum possible number of edges so that for each channel c, a message published for c should reach all subscribers of c by being forwarded by only the nodes subscribed to c. They prove that the corresponding decision problem is NP-complete, and present an approximation within a logarithmic ratio.

Scribe is a decentralized application-level multicast infrastructure implemented on top of Pastry, a peer-to-peer location and routing overlay on the Internet [CDKR02]. Each channel has an owner known to all nodes. Publications are first send to the channel's owner which distributes messages via a multicast tree to the channel's subscribers. Subscriptions to a channel are forwarded to the channel owner using Pastry's routing protocol. Nodes on a route snoop on subscribe messages. If the channel is one to which the current node already subscribes, it will stop forwarding the subscription and add the node as one of its children. This way a treelike routing structure is formed. Scribe relies on Pastry to optimize the routes from the owner to each subscriber. Fault tolerance is accomplished through the use of time-outs and heartbeat messages.

The characteristics of WSNs require light weight solutions. This excludes the above described approaches. This also holds for other solutions using complex routing structures such as Steiner-trees, rendezvous based, or informed gossiping.

There have been several efforts towards publish/subscribe systems for WSNs. Directed Diffusion is an early approach [IGE00]. Nodes issue subscriptions by broadcasting a query into the entire network. Upon receiving a query, each node creates a gradient entry in the routing table to point to the neighbouring node from which the query is received. Using a gradient path, matching messages are sent toward subscribers. An antipodal approach is followed by Huang et al. [HGM03]. Each publisher builds a broadcast tree to deliver messages to subscribers. Another approach is taken by the MQTT-S protocol [HTSC08]. Here, publication matching is carried out by a single central broker located on a node external to the WSN.

Mires is a publish/subscribe middleware implemented on top of TinyOS [SGV+05]. The subscription/publication protocols are similar to that of Directed Fusion. PS-QUASAR is a publish/subscribe middleware based on the Contiki operating system that handles Quality of Service support by means of multicasting techniques [CDRT13].

Fault tolerance of publish/subscribe systems has been addressed by various authors. Jerzak and Fetzer use a soft state approach [JF09]. Within the paradigm of self-stabilization this challenge has been tackled by Shen [She07] and Jaeger [Jae08] independently. The basic idea in the latter work is that routing entries are leased. All entries in the routing tables of all nodes must be periodically renewed, otherwise they are discarded from the routing table. The renewal of routing entries is triggered by the subscribers. A client must renew the lease for each of its subscriptions once in a refresh period. Shen proposes a system where all messages are routed along a single spanning tree T. Each node v holds a routing table consisting of a set of tuples of the form (c, R) where c is a subscription (to channel c), and R is a set of neighbouring nodes in T from which c was received. If a message arrives at v matching a subscription c in the routing table, then it is forwarded to all nodes in R, except for the node from which the message arrived. Subscriptions and unsubscriptions are also forwarded along T to all nodes. To maintain this routing, neighbouring nodes periodically exchange their routing tables. Inconsistencies between the tables lead to corrective actions. Nodes make local corrections independently and asynchronously. Through a sequence of local corrections the consistency among the distributed routing tables is eventually restored.

The drawback of the above described approaches is that messages mainly travel only along the tree edges. This can lead to unnecessary long paths and thus, to a high network load. Furthermore, Shen's approach lacks scalability mainly due to the periodic exchange of complete routing tables. We overcome these shortcomings by routing over shorter routes (not necessarily the shortest routes) and by employing the leasing technique.

6.3 GENERAL APPROACH

This work uses the asynchronous message-passing model where each node has a unique identifier. To cope with the asynchronous nature of the delivery of messages, queues are used. We assume that channels behave in a FIFO style. The system is fault tolerant with respect to transient faults. These can be caused by hardware errors, temporary unavailability of network links resulting in message duplication, loss, corruption, or insertion. Permanent faults and Byzantine behaviour are not considered. We assume that channel and node identifiers, along side with executable code, are stored in ROM which we expect never to be corrupted. Furthermore, we provide only non-masking fault tolerance. That is, after each transient fault the system will reach a legitimate state in finite time, hence, the service may be temporally unavailable. The legitimate status is kept until the next transient error.

6.3.1 *Routing of Publish/Subscribe Messages*

The task of a publish/subscribe system is to deliver all messages published into a channel to all clients that have subscribed to this channel until they unsubscribe from it. As usual we model the distributed system as an undirected graph $G = (V, E)$. Vertices or nodes in this graph are responsible for routing messages among each other. Jaeger [Jae08], for example, models a broker overlay network by G where each participating user is connected to a node of G. We dissociate this work from that assumption. Hence, the more common model in WSNs is used, were all nodes may be involved in the routing process, despite their occupation (e.g., publisher, subscriber, regular node).

In a publish/subscribe system only the subscribers are interested in published messages. Therefore, routing would be extremely effective if only subscribers for a certain channel would route messages. Obviously this is not achievable in general. To simplify the routing, we propose an overlay network structured as a ring. As most topologies do not contain a ring we suggest a virtual ring. Nodes can appear more than once on a virtual ring. We call the location of a node on the virtual ring their position. With our algorithm the ring has a total size of $2(n - 1)$ (where $n = |V|$) positions, that is, on average each node has two positions on the virtual ring. Routing is performed clockwise, in the sense of increasing positions (a designated node takes care of the wrap around). Messages are discarded (i.e., not forwarded) to, or over the originating position.

Scalability can not be achieved this way, therefore, *short-cuts* on the virtual ring are proposed. A short-cut skips positions on the ring to connect subscribers in a shorter way, while keeping in mind not to exclude subscribers of corresponding channels. Thus, each node maintains, for each of its positions and for each channel, the position of the neighbour that comes closest to the next subscriber for that particular channel in a table F. That means, that on average each node has to maintain two forwarding positions per node. After storing this information, routing becomes as simple as a single look-up in F. Keeping this table up-to-date with respect to

(un)subscriptions and transient faults is the main challenge of this approach. For the distributed algorithm, each node maintains two lists of positions: list P with its own and R with those of all neighbours.

6.3.2 *Example*

Before presenting the details of the implementation of the middleware we explain the main ideas using the example illustrated in Fig. 6.2. For simplicity we only consider a single channel in this example. The first sub-figure shows the network as seen by the MAC layer. The topology provided by the neighbourhood management layer Mahalle$^+$ is depicted in the second sub-figure for $C_N = 3$. Each node has at most three neighbours, dashed links are not used for communication in the layers above. As a consequence, e.g., only three of the six possible links of Node 9 are used. A larger value for C_N will result in more usable links at the cost of larger neighbour lists and larger routing tables at higher layers. A smaller value for C_N may result in a disconnected topology.

Based on this topology the spanning tree layer constructs the tree shown in Sub-figure 3, here Node 0 was selected as the root. Non-tree edges (depicted in gray) will later be used as short-cuts. The virtual ring layer computes the positions of each node using this spanning tree. The result with 30 positions on the ring, is shown in Sub-figure 4. In general there are $2(n-1)$ positions on a virtual ring. The fifth sub-figure shows the resulting virtual ring including all short-cuts as it will be used by the publish/subscribe system.

Next we discuss routing of publications for the example given in Sub-figure 6. At position 1 and 7 we have a single publisher (node 5) indicated by P. Furthermore, there are two subscribers, one at position 25 and 27 (node 11) and the other at 26 (node 7). Note, that the actual node identifier is transparent to the publish/subscribe layer. In Sub-figure 6, for all circled positions the routing table (current position *pos* and the message forwarding position *fPos*) is depicted as well, for other positions it is omitted for readability. Publishing always begins at the smallest position of a node, its home position. In this example it is position 1. Thus, the publication is forwarded over the closest short-cut to position 16. Since there exists no short-cut from 16, messages will be forwarded to 17. Routing proceeds in this style (i.e., without short-cuts) until the message reaches position 20. Now the forwarding position is 24. In the next step, position 25 - one of the subscribers - is reached, the publication is delivered, and the message will be forwarded to the next subscriber, i.e., 26. At 27 the message is discarded, because the *fPos* of position 27 is 13, which is bigger than the origin, i.e., position 1.

Additionally, the figure depicts the travelled path from P to the first S. The distance comprises 7 hops, even though this is much shorter than routing over the virtual ring in the regular way (24 hops), it is not the shortest path (4 hops).

FIGURE 6.2
Example topology with 16 nodes and C_N = 3 depicting the views of all layers and a generic routing scheme

6.4 ARCHITECTURE OF THE MIDDLEWARE

The self-stabilizing publish/subscribe middleware has a layered architecture. Each layer is self-stabilizing with respect to its service which is exposed by an API. The following sections describe the layers from top to bottom.

6.4.1 *Publish/Subscribe Layer*

The publish/subscribe layer provides the following API to be used by applications.

subscribe(c)	client subscribes to messages for channel c
unsubscribe(c)	client unsubscribes from channel c
publish(pub, c)	client publishes publication pub to channel c

Each node maintains a list C_S of the channels it holds subscriptions for. This list is made fault tolerant with respect to stale or corrupted entries with the lease technique. Each subscription is associated with a time-to-live (TTL) value that is periodically reset. If the value is not reset in time the entry is removed. In the following we sketch the implementation of two functions of this layer using the API of the layer below. The function ***sendOnRing()***provided by the virtual ring layer sends messages to the node located at the given position of the ring. Note that, function ***subscribe()***renews the subscriptions with period δ_S. If a subscription is not renewed in a time interval of length $2\delta_S$ it is discarded.

```
function PUBLISH(PUB, C)
    next := fwdPos(homePos, c);
    sendOnRing(next, PUB⟨homePos, pos, c, pub⟩);

function SUBSCRIBE(C)
    while subscribed do
        loop with period δ_S
            C_S.add(c);
            renew TTL for c with 2δ_S;
            /* Send message SUB to itself */
            sendOnRing(homePos, SUB⟨homePos, homePos, c, F[][0]⟩)
```

Publications are forwarded on the virtual ring. Message PUB⟨$pos, goal, c, pub$⟩ transports publication pub of channel c originating from position pos to position $goal$ (see Algorithm 10). If a receiving node is a subscriber of the corresponding channel, then the publication is delivered if it has not been delivered before. A publication is always delivered at the first position of a subscribing node reached while travelling the ring. A node can check this since it knows its own positions and the position of the point of origin of the publication.

By default publications are forwarded from one ring position to the next. The originating position is used to detect if a publication has travelled the complete ring, hence, that it can be discarded. To reduce latency and the number of messages short-cuts are used. For this purpose each node maintains for each channel c a table F which contains for each position of the node on the ring the next forwarding position. It is guaranteed that the omitted positions between $F[i][0]$ and $F[i][1]$ have no subscription for this channel. If the next forwarding position is beyond the originating position, the publication is discarded (see function ***isBetween()***).

To be fault-tolerant entries in F must be renewed periodically. For this purpose each entry has an associated TTL value which is decremented every time tick. If TTL = 0 for an entry x then the forwarding position $F[x][1]$ is set to $\perp$.

Subscriptions can be handled in two different ways. One possible routing scheme is to forward them on the virtual ring using the message $\text{SUB}\langle pos, goal, c, s_{list}\rangle$ (see Algorithm 10). The message is discarded when it reaches the originating position. Parameter s_{list} contains the set of positions on the ring of the subscribing node. Upon reception of a subscription a node must update the table F with the forwarding positions. For this purpose a node iterates over its own positions and checks whether a position of the new subscriber is closer to itself with respect to the current forwarding position. In this case the entry is updated and the TTL value is reset.

Another option is that a node forwards SUB messages to all its neighbouring positions. This may generate duplicated messages, as two messages sent to two different parts of the ring, may reach the same position after a certain time. To avoid this, the message is tagged with an end position. Once this position is reached the message is discarded. With this alternative subscription messages are forwarded concurrently. This may greatly reduce latency but also leads to more contention at the MAC layer.

6.4.2 *Virtual Ring Layer*

This layer provides and maintains a kind of overlay network in the form of a ring. It is only a virtual ring in the sense that a node may appear several times on this ring, adjoining positions correspond to neighbouring nodes. The positions on the ring are numbered beginning with Position 0. Each node maintains a list P with its own positions and a table R with all positions of all its neighbours sorted by positions for efficiency. The smallest position of each node is called the *home position*. The API of the virtual layer (shown below) enables the publish/subscribe layer to implement its functions in a self-stabilizing manner. It is implemented using P and R maintained with the help of the layer beneath.

There are several ways to construct virtual rings. We present a solution based on a rooted spanning tree. Each node has a variable *par* with the identifier of its parent in the tree and an array *chd* with the identifiers of its children. The virtual ring follows the path of a depth-first traversal of the tree. Instead of implementing such a traversal, we compute for each node the number of nodes in the subtree rooted at each child. With the help of these numbers it is straight forward to compute for

Algorithm 10 Handling of PUB and SUB messages

F	a table per channel, with each position *pos* of node v, forwarding position *fPos* for each *pos*, and a TTL
homePos	smallest of all positions of node v (i.e., $F[0][0]$)

function *FWDPos(POS, C)*

/*returns forwarding position for a pub. for channel *c* from position *pos**/

/*if no subscriber found return ***nextPosOnRing(***pos***)****/

p := **indexOf(***pos***)**][1];

if p = ⊥ **then**

return *nextPosOnRing(*pos***)***;

else

return F[**indexOf**(pos)][1];

Reception of: PUB⟨h, p, ch, $mess$⟩

if $ch \in C_S$ and *mess* not yet delivered to node **then**

deliver(*mess*);

f_p := ***fwdPos(***p, ch***)***;

if ¬***isBetween(***h, p, f_p***)*** **then**

sendOnRing(f_p, PUB⟨h, f_p,ch,$mess$⟩***)***

Reception of: SUB⟨h, p, ch, s_{list}⟩

/*$s_{list} = \{s_0, \ldots, s_k\}$: list of positions of (re)subscriptions from node *in**/

for i=0, …, **sizeOf**(F) **do**

find first s_j after $F[i][0]$ counter clockwise from s_{list};

P_{new} :=***getPosClosestTo(***$F[i][o]$, s_j***)***

if $F[i][1] = \perp$ || ***isBetween(***P_{new}, $F[i][o]$, $F[i][1]$***)*** **then**

$F[i][1] := P_{new}$;

renew TTL for p_i with $2\delta_S$;

f_p := ***nextPosOnRing(***p***)***;

if $f_p \neq h$ **then**

sendOnRing(f_p, SUB⟨h, f_p,ch,s_{list}⟩***)***

Algorithm 11 *

R	table with positions of all neighbours sorted by positions
P	list of own positions
$fresh_i$	boolean to indicate recently refreshed child count
cnt_i	number of children in sub-tree of child i

Variables and Functions of the Virtual Ring Layer

getIdAtPos*(pos)*	// returns *id* of neighbour at position *pos*
getPosClosestTo*(pos, goalPos)*	// returns largest counter clockwise position
	// after *pos* and before *goalPos* within R
nextPosOnRing*(pos)*	// return next ring position with wrap around
sendOnRing*(pos, msg)*	// send message *msg* to position *pos*
isBetween*(test, left, right)*	// checks if position *test* is in ccw ring segment
	// bounded by position *left* and *right*

each node its positions on the ring starting with Position 0 at the root. Note that, for this calculation the nodes do not need to know the total number of nodes. The computation of the positions is realized in two antidromic waves (see Algorithm 12). The first of these starts at the leaves. Recursively, each node tells its parent the number of nodes in its subtree. This is implemented with the message UP⟨*cnt*⟩. The second wave begins at the root and proceeds towards the leaves. A node that knows its home position and the size of the subtree rooted at each child can compute its own positions and the home positions of each child. This is implemented with the message DOWN⟨*cnt*⟩.

Fault tolerance is again achieved through a periodic repetition of this process. All leave nodes send message UP⟨1⟩ with period δ_R to their parents. Upon receiving such an UP message a node checks whether it has *fresh* (boolean indicating resent updates) values of the size of all sub-trees rooted at its children. If this is the case the node sends an UP message with the accumulated counts to its parent. If the root receives an UP message it starts the down wave by sending a DOWN message to all its children with their home position. Upon the reception of such a message a node sends a corresponding DOWN message to all its children.

6.4.3 *Spanning Tree Layer*

This layer maintains a spanning tree of the graph defined by the neighbourhood management protocol of the layer beneath. The implementation in Algorithm 13 is based on a self-stabilizing algorithm to construct a breadth-first tree of Huang and Chen adopted to the message passing model [HC92]. We assume the existence of a dedicated root node. In contrast to the other layers this layer uses broadcasts instead of point-to-point communication. A message TREE⟨*dist*, *parent*⟩ contains its current distance to the root and the identifier of its current parent. The root broadcasts with period δ_T the message TREE⟨0, *root*⟩. Upon the reception of

Algorithm 12 Virtual Ring

loop with period δ_R
  if $chd = \varnothing$ **then**
    send$(par, \text{UP}\langle 1\rangle)$

Reception of: UP$\langle cnt\rangle$ from s:
$cnt_s := cnt$
if $\forall i \in chd \mid fresh_i = true$ **then**
  for all $i \in chd$ **do**
    $fresh_i := false$
  if $\neg root$ **then**
    send$(par, \text{UP}\langle 1 + \Sigma_{chd} cnt_i\rangle)$
else
  $P[0] := 1$
  for all $i \in chd$ **do**
    $P[i] := P[i-1] + 2 \times cnt_i$
    send$(chd_i, \text{DOWN}\langle P[i-1]\rangle)$

Reception of: DOWN$\langle pos\rangle$ from s:
$P[0] := pos + 1$
for all $i \in chd$ **do**
  $P[i] := P[i-1] + 2 \times cnt_i$
  send$(chd_i, \text{DOWN}\langle P[i-1]\rangle)$

a message TREE$\langle d, p\rangle$ not originating from a current child a node checks whether it has to update the variables *par* and *dist*. If this is the case then it broadcasts the new values of these variables with a TREE message. If the message indicates that the sender considers the receiving node as its parent the node adds the sender to its set of children and resets the flag $upToDate$. Within an interval of length $2\delta_T$ the set *chd* is checked for stale entries, i.e., for which $upToDate = false$. These entries are removed. For the remaining entries the flag is set to $false$.

To reduce traffic, the periodically broadcasted messages at the spanning tree layer are used to inform a node's neighbours about the node's position on the virtual ring, as well. For this purpose every node appends its current positions to the broadcasted TREE message and receiving nodes use this information to update their table R. This cross layer approach is only an option. Alternatively, the positions can also be broadcasted on the virtual ring layer, ensuring the separation of layers while increasing the network load.

6.4.4 *Neighbourhood Management and MAC Layer*

This layer consists of the algorithm presented in [STW+13]. It provides a connected stable topology for WSNs by choosing well suited links among its neighbours. The number of neighbours of a node is limited to C_N. For the MAC-Layer we consider any protocol which supports point-to-point and broadcast communication patterns (e.g., IEEE 802.15.4).

Algorithm 13 Spanning Tree

chd	set of children of node v
$upToDate_i$	boolean to indicate recently refreshed tree data
par	parent of node v
$dist$	distance to root

loop with period δ_T
  if *root* **then**
    $dist := 0$
    broadcast(TREE$\langle dist, par\rangle$)

---

loop with period $2\delta_T$
  for all $i \in chd$ **do**
    if $upToDate_i = false$ **then**
      $chd.remove(i)$
    else
      $upToDate_i := false$

Reception of: TREE$\langle d, p\rangle$ from s:
if $p \neq id$ **then**
  if $d + 1 < dist$ **then**
    $dist := d + 1$
    $par := p$
    broadcast(TREE$\langle dist, par\rangle$)
  if $d + 1 = dist \wedge p < par$ **then**
    $par := p$
else
  $chd.add(s)$
  $upToDate_{chd=s} := true$
  broadcast(TREE$\langle dist, par\rangle$)

6.5 ANALYSIS OF ALGORITHMS

Following the detailed description of our approach, a discussion of its main properties is presented. The aim is to put the algorithms performance and requirements in perspective to the constrains of WSNs.

6.5.1 *Space requirements and scaling*

Each node has at most C_N neighbours and each neighbour occupies at most C_N positions on the ring. Thus, tables R and P together require $O(C_N^2)$ memory. Per channel, table F requires $O(C_N)$ memory. All other variables together require $O(C_N)$ memory. This leads to a total memory requirement of $O(C_N^2 + cC_N)$ (c denotes the number of channels). Thus, our approach scales with the size of the network, but not that well with the number of channels.

There are $2(n-1)$ positions on the virtual ring thus, a subscription periodically generates $2(n-1)$ messages. The quantity of messages per publication depends on the number of subscribers and on the structure of the network, this makes an analysis rather difficult. We conjecture that the number of messages scales well with the size of the network.

Next we prove that Algorithm 10 is correct, meaning that it eventually yields a correct routing of the messages as described by the following safety properties:

($\boldsymbol{P}_1$) A message is delivered to a node at most once.

($\boldsymbol{P}_2$) A node only receives messages belonging to a channel it has subscribed for.

($\boldsymbol{P}_3$) After subscribing to a channel a client eventually receives all messages published to that channel.

THEOREM 13. *As long as no transient fault occurs Algorithm 10 will eventually satisfy safety properties $\mathcal{P}_1$ to $\mathcal{P}_3$.*

Sketch of Proof of 13. Consider a point in time after which no transient error occurs, this includes events which lead to changes in the neighbourhood relation. Then eventually Algorithm 13 stabilize. This can be seen by adopting the original proof of [HC92] to our model. Note that, in contrast to this algorithm in our case a node does not need to know the size of the network. The reason is that the root periodically broadcasts TREE messages. The leasing technique is also used to correct faults in the set of children of a node. Thus, these sets are also eventually correct and stable. Once Algorithm 13 has stabilized also Algorithm 12 will stabilize. This is because leaf nodes of the spanning tree periodically send UP messages to their parents. These prompt the parents to send UP messages towards the root. Upon receiving an UP message the root sends DOWN messages to all its children which are then forwarded towards the leaf nodes. Hence, table P at all nodes will eventually be correct. In addition, this also holds for table R. Because a node periodically broadcasts its table P. Note that this table has at most C_N entries.

When a node receives a PUB messageand detects that the next forwarding position on the ring is beyond the originators position, then the PUB messagewill be discarded. Thus, a PUB messagenever cycles forever. A publication may reach a node at several positions, nevertheless it is delivered only at the first position reached. Since, eventually the set C_S is correct for each node, property $\mathcal{P}_1$ will be satisfied. After C_S is correct property $\mathcal{P}_2$ is guaranteed. To prove property $\mathcal{P}_3$, note that, after $O(n)$ time following a subscription to a channel by a node the corresponding data structure F is correctly updated at every node. After this point in time the node receives all publications to this channel. 13

6.5.2 *Timings and time-outs*

The self-stabilization property of the presented middleware heavily relies on the leasing technique. A critical issue with this method are the values for the time-outs. With large values a corrupted or lost entry (e.g., a subscription) may remain undetected longer than can be tolerated by the application. For example a corrupted entry in table F may lead to a situation where a subscriber will miss some publications. The larger the time-out value the higher the probability for this case. Smaller time-outs may generate unnecessary overhead, in particular a high number of messages. In case of a fault arising from lost messages, short time-outs may aggravate the situation, because they place a higher load on a potentially already overloaded channel.

The middleware uses time-outs on each layer: δ_S to renew subscriptions, δ_R to renew the positions on the virtual ring, and δ_T to renew the spanning tree. The actual values leading to a stable behaviour strongly depend on the characteristics of the network and the frequency of faults. Even expressing constraints about the relation between time-out values turns out to be difficult. The fact that the virtual ring is completely determined by the spanning tree suggests that the value of δ_R should be larger than δ_T. This is because faults in the tree are only repaired after a period of length δ_T. But on the other hand, repairing corruptions in the table R are independent of the spanning tree. Thus, it also makes sense to choose δ_R smaller than δ_T. We believe that the values of time-outs should be determined adaptively at runtime.

6.5.3 *Overview of simulation results*

We implemented and simulated our solution using the OMNeT++framework, together with the MiXiMextension it offers realistic path-propagation models to mimic WSN behaviour. In simulation the neighbourhood management layer was able to create a stable topology. Nonetheless, there are lower bounds for the desired C_N by the neighbourhood algorithm. A value for C_N permitting too few neighbours will lead to a very long stabilizing process and may not create a connected setup. Especially in large networks with 100 or more nodes, less then, e.g., 6 neighbours will put high difficulty on the neighbourhood protocol. Therefore, we used $C_N = 10$ which still recognizes the restricted memory of sensor nodes. Changes in the neighbourhood of a node will change the tree setup, hence, the ring positions, this will in fact lead to errors during the routing of subscriptions and publications. MiXiM's log normal shadowing, path-loss models, and bit-corruption mechanisms where used to induce link changes, i.e., possible errors. To mitigate the problem of too many messages being used for re-submission, ring maintenance, tree building, and neighbourhood management we use the trickle algorithm [LCH+11]. It adapts the update speed of information depending on the changes in message content.

Only a brief report on simulation results is given due to space restrictions. Figure 6.3 shows our solution compared to a flooding approach which was implemented and tested using the same underlying network (grid network, nodes in communication range up to 20, varying number of subscribers and publishers randomly distributed, multiple runs, 7 differed networks sizes).

The delivery rate for both approaches declines with growing network size. Depending on the stability of our proposed layered structure high delivery rates can be achieved even when the network size increases. The delivery rate declines as the probability for instabilities accumulates. The needed messages, i.e., the power consumption of our approach depends very much on the ratio of publication messages to system maintenance messages. For the simulations depicted in Figure 6.3 we have sent at least 3 publications for every (re-)subscription message, and with the trickle timer we could decrease other maintenance messages as well. Furthermore, with growing numbers of subscribers more messages need to be sent. In case of flooding

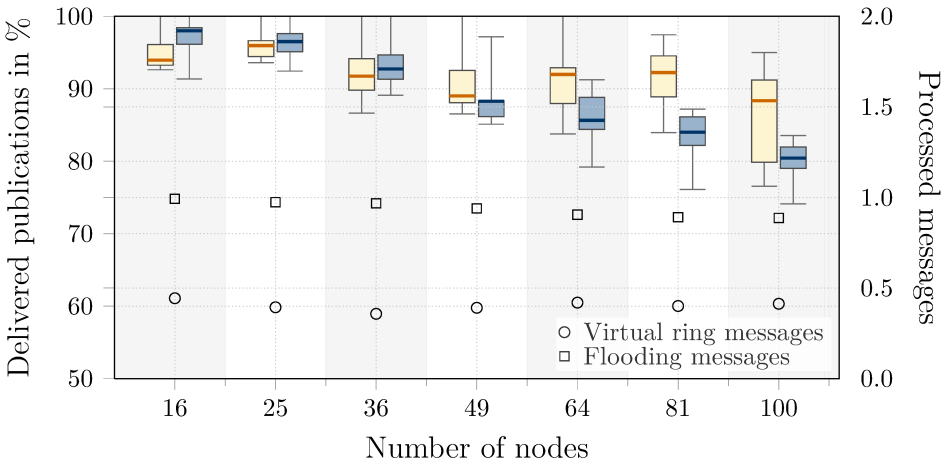


FIGURE 6.3
Message delivery of flooding (dark box) compared to the virtual ring approach (light box). Processed messages normalized over number of nodes and publishers.

each node generates one message per publication. This holds for any number of subscribers, unless the message does not reach all nodes due to collisions.

6.6 CONCLUSION AND OUTLOOK

We presented a scalable, self-stabilizing middleware for channel-based publish/subscribe that particularly meets the requirements of WSNs. We consider message and memory corruptions while respecting dynamic network changes, such as, node and link removals and additions. The middleware aims to capture the trade-off between the scalability of the overlay network (i.e., the size of the routing tables) and the message routing overhead incurred by nodes forwarding publications. As optimal solutions (e.g., Steiner tree based overlays) are too costly to build and maintain, we use a simpler structure of a virtual ring. To reduce message complexity we use channel specific short-cuts on this ring. With the help of a neighbourhood management protocol the system suffices with an average storage demand per node that is only proportional to the number of channels, i.e., independent of the network size.

The middleware is organized as four independent layers. Each layer is self-stabilizing on its own. For their composition we use collateral composition. Fault tolerance is mainly achieved through the leasing technique. Evaluation using simulation showed that the system's performance highly depends on the used neighbourhood management protocol. If a high stability can be preserved, then the middleware quickly and effectively routes publications to all subscribers, while the messaging demand per node stays constant for the maintenance structure. An open issue is the lack of scalability with respect to channels in general. Here we hope to adopt

concepts from [CMTV07].

Conclusion and future directions

7

In this thesis we focused on designing efficient self-stabilizing algorithms. This was mainly done for matching problems in the first part of the document and then for publish/subscribe systems in its second. This work was addressed in the setting of the state model and the link-register for the matching problems and the message passing model for publish/subscribe systems. This choice of models is mainly motivated by the validation strategy used in both cases. For the matching part, algorithms are proved, and hence an abstract model was needed. It is also the reference model for comparing the different results obtained for this problem. The algorithm for the publish/subscribe paradigm is implemented for a simulation, and thus required a more detailed model, that can be expressed through modern day programming languages routines. The summary of the different results obtained are to be detailed according to this two part scheme.

7.1 MATCHING PROBLEMS

From the self-stabilizing maximal matching algorithm in identified networks, presented in Chapter 3, we derived related problems by relaxing some hypotheses or adding some constraints to it. The succession of the results in this part can be thought of as trying to design a polynomial self-stabilizing algorithm for these different cases. The obtained results are valid for the distributed adversarial daemon.

7.1.1 *Maximal Matching in anonymous networks*

We give a polynomial probabilistic self-stabilizing algorithm for the maximal matching in anonymous networks. Due to the anonymity constraint, the algorithm is randomized to break symmetry. It stabilizes in $O(n^2)$ moves with high probability and improves on the last known result that stabilizes in $O(mn^3)$ expected moves.

FUTURE DIRECTIONS For now, it remains an open question whether the same order of complexity can be obtained in the same setting (daemon and anonymity) for any approximation of the maximum matching problem better than 1/2, which is the maximal matching. This seems difficult since the next approximation is a

2/3–approximation that can be achieved by finding and exploiting 3-augmenting paths. This requires being able to distinguish a path from a cycle, which is impossible in a deterministic way in anonymous networks.

7.1.2 *A 2/3–approximation of the maximum matching problem in identified networks*

In this work [CMMP16] a proof that the best known self-stabilizing 2/3–approximation of the maximum matching stabilizes in $O(2^{\sqrt{N}})$ moves is given by exhibiting an execution that has needs exactly this number of moves to recover from a fault. The tightness of this exponential bound have, until now, not been proven. Following this, we present the first polynomial 2/3–approximation of the maximum matching problem in identified networks and under the distributed adversarial daemon. It stabilizes in $O(n^2)$ moves, provided that an underlying maximal matching has been initially built. The overall, from scratch, complexity can be raised to $O(m \times n^2)$ moves through fair composition.

FUTURE DIRECTIONS It is hard to see this straightforwardly extend to the case of weighted matchings, as information look-up over a path of length three is not sufficient. It is also interesting to think of the matching problem as a partitioning of the graph into cliques of size two. From here, it is natural to ask whether it is possible to obtain a partition into cliques of larger size under the same conditions. The most recent work [DLR14] tackling this question requires a spanning tree structure for the graph and a leader election mechanism. Its overall complexity is roughly $O(n^{11})$. It is an open question to know whether it is possible to achieve it with less cost or without the aforementioned hypotheses.

7.2 SELF-STABILIZING PUBLISH/SUBSCRIBE SYSTEMS

We presented a solution for routing messages in channel-based publish/subscribe systems. This solution adopts a spanning tree as a virtual ring mechanism and introduces for the first time the notion of short-cuts for routing messages outside the virtual ring when it is more efficient to do so. It is also more space efficient by not requiring every node of the system to store a large routing table. The algorithm remains also valid in the case of wireless sensor networks, where the topology is highly dynamic by adopting results from neighbourhood management protocols.

FUTURE DIRECTIONS It remains open whether we can derive theoretical values for the different time-outs used for the leasing technique. On the other part, it is interesting to tend toward a smaller virtual ring structure than the spanning tree

and a less costly one than the Steiner Tree. An approximation of the Steiner Tree as in [BPBR09] could be considered. Special spanning trees with bounded average degrees could also reduce the time a message has to travel on the ring.

Figure 6.3 Message delivery of flooding (dark box) compared to the virtual ring approach (light box). Processed messages normalized over number of nodes and publishers. 104

List of Algorithms

Bibliography

[AGM+11] Heiner Ackermann, Paul W Goldberg, Vahab S Mirrokni, Heiko Röglin, and Berthold Vöcking. Uncoordinated two-sided matching markets. *SIAM Journal on Computing*, 40(1):92–106, 2011.

[AI15] Y. Asada and M. Inoue. An efficient silent self-stabilizing algorithm for 1-maximal matching in anonymous networks. In *WALCOM: Algorithms and Computation - 9th International Workshop*, pages 187–198. Springer International Publishing, 2015.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[Bal85] Alexandru T Balaban. Applications of graph theory in chemistry. *Journal of chemical information and computer sciences*, 25(3):334–343, 1985.

[BFM08] P. Berenbrink, T. Friedetzky, and R. A. Martin. On the stability of dynamic diffusion load balancing. *Algorithmica*, 50(3):329–350, 2008.

[BPBR09] Lélia Blin, Maria Gradinariu Potop-Butucaru, and Stephane Rovedakis. A superstabilizing log(n)-approximation algorithm for dynamic steiner trees. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '09, pages 133–148, Berlin, Heidelberg, 2009. Springer-Verlag.

[CDKR02] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony I. T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE J. Sel. Areas in Com.*, 20(8):1489–1499, 2002.

[CDRT13] Jaime Chen, Manuel Díaz, Bartolomé Rubio, and José M. Troya. PS-QUASAR: A publish/subscribe QoS aware middleware for Wireless Sensor and Actor Networks. *J. of Sys. & Softw.*, 86(6):1650–1662, 2013.

[CHS02] S. Chattopadhyay, L. Higham, and K. Seyffarth. Dynamic and self-stabilizing distributed matching. In *Proceedings of PODC*, pages 290–297. ACM, 2002.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[CLM+16] Johanne Cohen, Jonas Lefevre, Khaled Maâmra, Laurence Pilard, and Devan Sohier. A self-stabilizing algorithm for maximal matching in anonymous networks. *Parallel Processing Letters*, 26(04):1650016, 2016.

[CMMP16] J. Cohen, K. Maâmra, G. Manoussakis, and L. Pilard. Polynomial self-stabilizing maximum matching algorithm with approximation ratio 2/3. In *International Conference On Principles Of Distributed Systems, OPODIS*, 2016.

[CMTV07] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Constructing scalable overlays for pub-sub with many topics. In *Proc. 26th Annual ACM Symp. on Prin. of Distributed Computing*, pages 109–118, 2007.

[CRW01] Antonio Carzaniga, David Rosenblum, and Alexander Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comp. Syst.*, 19(3):332–383, 2001.

[DH03] D. E. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Inf. Process. Lett.*, 85(4):211–213, 2003.

[Die] Reinhard Diestel. *Graph Theory*. Fourth edition.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[DIM89] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems. In *Proceedings of the MCC Workshop on Self-stabilizing Systems*, 1989.

[DLM16] Ajoy K Datta, Lawrence L Larmore, and Toshimitsu Masuzawa. Maximum matching for anonymous trees with constant space per process. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 46. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[DLR14] François Delbot, Christian Laforest, and Stephane Rovedakis. Self-stabilizing algorithms for connected vertex cover and clique decomposition problems. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, pages 307–322, 2014.

[Dol00] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[DT11] Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. *CoRR*, abs/1110.0334, 2011.

[Edm87] Jack Edmonds. *Paths, Trees, and Flowers*, pages 361–379. Birkhäuser Boston, Boston, MA, 1987.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[EPSW14] Y. Emek, C. Pfister, J. Seidel, and R. Wattenhofer. Anonymous networks: randomization = 2-hop coloring. In *Proceedings of PODC*, pages 96–105. ACM, 2014.

[Gal86] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18(1):23–38, March 1986.

[GC89] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.*, 23(5):202–210, 1989.

[GHJS08] W. Goddard, S. T. Hedetniemi, D. Pokrass Jacobs, and P. K. Srimani. Anonymous daemon conversion in self-stabilizing algorithms by randomization in constant space. In *Int. Conf. in Distr. Computing and Networking*, volume 4904, pages 182–190, 2008.

[GHS06] W. Goddard, S. T. Hedetniemi, and Z. Shi. An anonymous self-stabilizing algorithm for 1-maximal matching in trees. In *Proceedings of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 797–803, 2006.

[GJ01] M. Gradinariu and C. Johnen. Self-stabilizing neighborhood unique naming under unfair scheduler. In *7th Int. Euro-Par Conference*, volume 2150, pages 458–465, 2001.

[GK10] N. Guellati and H. Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *J. Parallel Distrib. Comput.*, 70(4):406–415, 2010.

[GM96] B. Ghosh and S. Muthukrishnan. Dynamic load balancing by random matchings. *J. Comput. Syst. Sci.*, 53(3):357–370, 1996.

[HC92] Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Inf. Process. Lett.*, 41(2):109–117, 1992.

[HGM03] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe tree construction in wireless ad-hoc networks. In *Mobile Data Management*, volume 2574 of *LNCS*, pages 122–140. Springer, 2003.

[HH92] S.-C. Hsu and S.-T. Huang. A self-stabilizing algorithm for maximal matching. *Inf. Process. Lett.*, 43(2):77–81, 1992.

[HJS01] S. T. Hedetniemi, D. Pokrass Jacobs, and P. K. Srimani. Maximal matching stabilizes in time $O(m)$. *Inf. Process. Lett.*, 80(5):221–223, 2001.

[HK73] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[Hoe13] M. Hoefer. Local matching dynamics in social networks. *Inf. Comput.*, 222:20–35, 2013.

[HTSC08] U. Hunkeler, Hong Linh Truong, and A. Stanford-Clark. MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks. In *3rd Int. Conf. on Com. Systems Soft. & Middleware*, pages 791–798, Jan 2008.

[IGE00] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. 6th Int. Conf. on Mobile Comp. & Netw.*, pages 56–67, 2000.

[IOT16] Michiko Inoue, Fukuhito Ooshita, and Sébastien Tixeuil. An efficient silent self-stabilizing 1-maximal matching algorithm under distributed daemon without global identifiers. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 195–212. Springer, 2016.

[Jae08] Michael A. Jaeger. *Self-Managing Publish/Subscribe Systems*. PhD thesis, Techn. Univ. Berlin, 2008.

[JF09] Zbigniew Jerzak and Christof Fetzer. Soft state in publish/subscribe. In *Proc. 3rd ACM Int. Conf. on Distributed Event-Based Systems*, pages 17:1–17:12, 2009.

[Kle00] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. pages 163–170, 2000.

[Knu76] D. Knuth. *Marriages stables et leurs relations avec d'autres problèmes combinatoires*. Les Presses de l'Université de Montréal, 1976.

[KS00] Mehmet Hakan Karaata and Kassem Afif Saleh. Distributed self-stabilizing algorithm for finding maximum matching. *Comput Syst Sci Eng*, 15(3):175–180, 2000.

[Lam85] Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. volume 19, pages 34–44, New York, NY, USA, October 1985. ACM.

[Lan] Serge Lang. *Algebra*. Third edition.

[LCH+11] Philip Levis, T Clausen, J Hui, O Gnawali, and J Ko. The trickle algorithm. *Internet Engineering Task Force, RFC6206*, 2011.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[MB09] D König Michael and Stefano Battiston. From graph theory to models of economic networks. a tutorial. *Networks, Topology and Dynamics*, pages 23–63, 2009.

[MM07] F. Manne and M. Mjelde. A self-stabilizing weighted matching algorithm. In *9th Int. Symposium Stabilization, Safety, and Security of Distributed Systems (SSS)*, Lecture Notes in Computer Science, pages 383–393. Springer, 2007.

[MMPT09] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science (TCS)*, 410(14):1336–1345, 2009.

[MMPT11] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. *Theoretical Computer Science (TCS)*, 412(40):5515–5526, 2011.

[Pre99] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *16th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, Lecture Notes in Computer Science, pages 259–269. Springer, 1999.

[SCWB08] David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4):615–633, Dec 2008.

[SGH04] Z. Shi, W. Goddard, and S. T. Hedetniemi. An anonymous self-stabilizing algorithm for 1-maximal independent set in trees. *Inf. Process. Lett.*, 91(2):77–83, 2004.

[SGV+05] Eduardo Souto, Germano Guimares, Glauco Vasconcelos, Mardoqueu Vieira, Nelson Rosa, Carlos Ferraz, and Judith Kelner. Mires: A publish/subscribe middleware for sensor networks. *Personal Ubi. Comput.*, 10(1):37–44, 2005.

[She07] Zhenhui Shen. *Techniques for building a scalable and reliable distributed content-based publish/subscribe system*. PhD thesis, Iowa State Uni., 2007.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[SMT14] Garry Siegemund, Khaled Maâmra, and Volker Turau. Brief announcement: Publish/subscribe on virtual rings. In *Proc.* 16^{th} *SSS*, 2014.

[STM15] Gerry Siegemund, Volker Turau, and Khaled Maamra. A self-stabilizing publish/subscribe middleware for wireless sensor networks. In *2015 International Conference and Workshops on Networked Systems, NetSys 2015, Cottbus, Germany, March 9-12, 2015*, pages 1–8, 2015.

[STW+13] Gerry Siegemund, Volker Turau, Christoph Weyer, Stefan Lobs, and Jörg Nolte. Brief Announcement: Agile and Stable Neighborhood Protocol for WSNs. In *Proc. 15th SSS*, volume 8255 of *LNCS*, 2013.

[SUW15] J. Seidel, J. Uitto, and R. Wattenhofer. Randomness vs. time in anonymous networks. In *DISC*, volume 9363 of *LNCS*, pages 263–275. Springer, 2015.

[Tel01] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2001.

[TH11a] V. Turau and B. Hauck. A fault-containing self-stabilizing (3 - 2/(δ+1))-approximation algorithm for vertex cover in anonymous networks. *Theoretical Computer Sciences*, 412(33):4361–4371, 2011.

[TH11b] V. Turau and B. Hauck. A new analysis of a self-stabilizing maximum weight matching algorithm with approximation ratio 2. *Theoretical Computer Science (TCS)*, 412(40):5527–5540, 2011.

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Algorithmes auto-stabilisants efficaces pour les graphes

Mots clés : Auto-stabilisation, Théorie des graphes, Algorithmique distribuée.

Résumé : L'auto-stabilisation est un domaine de l'algorithmique qui vise la conception d'algorithmes distribués qui convergent vers une solution même après une corruption d'un ou plusieurs agents. Les graphes offrent un modèle de topologie pour l'execution de ces algorithmes. Dès lors, il est naturel de se demander s'il existe des algorithmes auto-stabilisants efficaces qui calculent certaines de leurs propriétés. Le problème du couplage est en particulier intéressant, il s'agit de partitioner le graphe en un maximum d'arêtes disjointes. Ce problème a été étudié dans le cadre auto-stabilisant et des algorithmes comme celui de Manne et al ont été élaborés. L'objectif de cette thèse est l'amélioration de ces algorithmes en complexité ainsi que d'en concevoir pour d'autres variantes du problème.

L'apport majeur de cette thèse est un travail en deux parties.
Il a d'abord été conduit pour le couplage maximal, dans sa version anonyme, résultant en un algorithme auto-stabilisant probabiliste, de complexité polynomiale avec forte probabilité.
Il offre aussi un outil pour la transformation d'algorithmes identifiés en algorithmes anonymes.
Pour la version maximum du couplage, nous donnons une preuve que la borne sub-exponentielle de l'algorithme pour une 2/3-approximation de Manne et al est atteinte. Cette borne était jusque là supposée grossière. Nous donnons ensuite un algorithme polynomial pour ce problème de 2/3 approximation. Les deux algorithmes reposent sur un 1-couplage en utilisant le théorème de Hopcroft et Karp.
La seconde partie porte sur l'élaboration d'un algorithme auto-stabilisant pour les systèmes Pub/Sub basés sur une communication en canal. Le résultat de ces travaux est un algorithme auto-stabilisant dynamique pour le routage des messages dans un tel système. Il s'appuie sur une composition de couches et sur une structure d'anneau virtuel. Il introduit la nouvelle notion de Raccourci dans ce type d'anneaux permettant de disposer des messages d'annonce ainsi que ceux de désinscription. Ceci permet d'atteindre une réduction du temps de routage des messages, ainsi que de l'espace requis par chaque noeud du graphe.

Title : Efficient self-stabilizing algorithms for graphs.

Keywords : Self-stabilization, Graph theory, Distributed algorithms.

Abstract : Self-stabilization is a field of algorithms design that focuses on designing distributed algorithms that converge toward a legitimate behavior in the presence of transient faults. As for other distributed algorithms, graphs provide a topological model for their execution. In this context, it is naturel to ask for efficient self-stabilizing algorithms computing different properties of these graphs. The matching problem in particular, is about portioning the graph into the maximum number of disjoint couples. This problem has been studied in the literature and algorithms for solving it were designed, the Manne et al algorithm is one of them and the most recent. The main goal of this thesis is the improvement, with respect to complexity, of these algorithms and to design new ones for other variants of the problem.

The contribution of this thesis is structured in two parts.
First, a self-stabilizing algorithm for the maximal matching problem in anonymous graphs. This algorithm is randomized and reaches a solution with high probability in polynomial time improving upon t
For the maximum version of the problem, a polynomial self-stabilizing algorithm computing a 2/3 approximation is designed. This algorithm improves on the Manne et al algorithm, as we prove that it is sub-exponential. This bound was conjectured, until now, not to be tight.
The second part focuses on topic-based Pub/Sub systems. These are distributed systems where information from a set of publishing nodes has to reach a set of subscribers according to different topics. A dynamic self-stabilizing algorithm for routing information on such systems is given. It is based on a composition technique offering a virtual ring upon which the new notion of short-cuts has been introduced. This new algorithm does not rely on un-subscription or announcement messages reducing the routing time as well as the space required by each node of the graph.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France