Chapter 2

Self-Stabilization

This chapter provides an introduction to self-stabilizing algorithms and related work. The first section describes conventional distributed algorithms and the models of computation. Section 2.2 starts with the categorization of faults in distributed systems and fault tolerance. It introduces self-stabilization and gives a more formal definition of the terms and concepts used in this thesis. Several methods to measure the complexity of self-stabilizing algorithms are discussed. Section 2.3 presents methods to design a self-stabilizing algorithm. Finally, Section 2.4 provides an overview of self-stabilizing algorithms for classical graph problems. More related work on specific problems can be found in the corresponding chapters.

2.1 Distributed Algorithms

In the literature, different definitions for the term *distributed system* can be found. Tanenbaum and van Steen [TS06] provide a definition that emphasizes the transparency property:

> A distributed system is a collection of independent computers that appears to its users as a single coherent system.

A famous aphorism by Lamport [Lam87] alludes to this property:

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

Bal et al. [BST89] characterize a distributed system in a more technical manner:

> A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communication network.

They also discuss the disagreement on the term "distributed system" in the literature [BST89]. The definition of Bal et al. will be used throughout this thesis with the understanding that this definition is not limited to physical processors but also considers other autonomous units or *nodes* such as processes. The latter is what Bal et al. call a *logically distributed software system*, but their distinction is not needed on the level of abstraction of this thesis. The communication network mentioned in the definition is considered to be a (connected) graph and only adjacent nodes can communicate with each other directly.

Two main models of distributed systems are distinguished in the literature [Pel00]: The synchronous model and the asynchronous model, the difference being whether there are upper bounds on the time certain processes are allowed to consume. The asynchronous model does not make any assumptions on the duration of a computational step or message delay, apart from being finite. Thus, messages that are sent but not received within a certain time cannot be considered to be lost but may be received later. On the other hand, the synchronous model assumes fixed time intervals for computations and guarantees that any message is received within a given time (which is known to all nodes). Hence, an advantage of synchronous systems is that lost messages can be detected. In this thesis the degree of synchrony of the distributed system is determined by the model used for the *atomicity of communication* and the assumed *scheduler*. These terms will be explained later. More detailed information about distributed systems in general can be found e.g. in [CDK05].

A *distributed algorithm* is an algorithm specifically designed to run in a distributed system. The nodes can operate concurrently and they communicate with each other to achieve a common goal. The most significant difference compared to conventional algorithms is the lack of a central entity that has access to the global state, i.e. the state of each node. All nodes act autonomously and the basis for their decisions is local knowledge only: The nodes hold their own state and can retrieve the state of their neighbors.

It is possible to gather the local state of all nodes by passing a neighbor's state on to the next node until some special node has aggregated the information of the whole system and can send tasks to the other nodes, but that is contrary to the idea of a distributed algorithm. Furthermore this procedure requires time and memory proportional to the size of the graph. The same arguments hold for a similar approach: If all nodes determine the topology of the whole distributed system, they can calculate their final state locally via the execution of an algorithm that is not restricted to local knowledge.

Having a "distributed state" and nodes that execute their algorithm according to local information only, different parts of the system may temporarily veer away from their common goal without knowing it. This also depends on the *locality* of the given problem or algorithm, i.e. to which extent the state of a node far away from a certain node influences its own state. An example for such a dependency is given in Section 4.2. More information about the locality of specific problems can be found in [NS95, MNS95, AGLP89, Suo11].

The atomicity of communication between the nodes can be modeled in miscellaneous ways for distributed algorithms [AW04, Tix09]. Tixeuil [Tix09] emphasizes that most literature in the context of self-stabilizing algorithms uses a high level of atomicity and lists the three most common models:

1. The *state model* (or *shared-memory model with composite atomicity*, [Dij74, Dol00]): In this model, reading the states of all adjacent nodes and updating its own state is considered an atomic action.

2. The *shared-register model* (or *read-write atomicity model*, [DIM93]): This model treats a single read and a single write operation as atomic actions. This model is the more general one, but there are methods for transforming algorithms from one model to the other [Dol00].

3. The *message-passing model* [AB93, DIM97a, KP90]: Here, an atomic step consists of either sending a message to one of the neighboring nodes, or receiving such a message.

The latter model requires to explicitly use the send and receive operation in an algorithm to exchange messages. The first two models simulate a common memory area for two adjacent nodes. In these cases, lower layers realize the information

exchange [Tel01]. Where not explicitly stated otherwise, this thesis assumes the state model for the algorithms. Another model is often used for algorithms in anonymous networks (see below):

4. The *link-register model* with composite atomicity [DIM93]: In this model, a node uses two separate registers for each neighbor (a read and a write register), i.e. a node can only read "its own" segment of its neighbors memory. Reading its registers from all neighbors and updating its own registers is considered one atomic operation. A more formal introduction to the link-register model is provided in Chapter 5.

Distributed algorithms substantially depend on the properties of the underlying network. In a *uniform* network all nodes execute the same algorithm. Non-uniform networks allow the nodes to execute distinct algorithms. A very important property is the availability of a symmetry-breaking mechanism. Such a mechanism is needed e.g. if it is undesirable that two adjacent nodes change their state at the same time. The most common model assumes all nodes to have unique identifiers. These can be used to ensure local mutual exclusion. For instance, in [GT07] the nodes have to set a boolean flag to tell their neighbors in advance when they want to change their state. A node is allowed to change its state only if none of its neighbors with smaller identifier has also set its flag.

Non-uniform networks can use another mechanism to break the symmetry by having a node that takes on a special role. These two network models are equivalent [Dol00]. In uniform networks without unique identifiers it is possible to use randomization to break symmetry. Availing oneself of randomization results in a probabilistic algorithm, though. A network is called *anonymous* if it is uniform and there are no further symmetry breaking mechanisms such as unique identifiers or randomization.

A lot of research has been done in the field of algorithms in anonymous networks. Angluin made the most remarkable publication in that area by proving several impossibility results subject to the different anonymity properties of the network [Ang80]. In particular Angluin showed that it is impossible to break symmetry via a *port numbering* (i.e., an edge ordering, for details see Chapter 5) in general graphs. Most of the algorithms in this thesis assume a uniform network and that all nodes have (locally) unique identifiers. Only in Chapter 5 an anonymous network is assumed.

2.2 Fault Tolerance and Self-Stabilization

In general, it is impossible to guarantee that a system will stay free of faults all the time. Hence, there must be a strategy to handle errors if they occur. Conventional systems may have a central unit that detects errors and decides which measures have to be taken. In a distributed system, error-handling is inherently more difficult: The detection of an error is not as simple due to the lack of a node with global knowledge, and also the nodes have to cooperate and coordinate their actions in order to overcome the erroneous state. Furthermore, there are types of errors that occur more likely in a distributed system. For instance, in a wireless sensor network a node can fail due to a depleted battery or physical damage. Messages can get lost, they may be duplicated or arrive in a different order.

Apart from errors there are other scenarios that can make a distributed system end up in an illegitimate state, e.g. if new nodes are added to the system or some nodes are removed from it. Locating the source of an error, replacing or removing an erroneous node, or permanently monitoring the whole system to detect faults and perform a global reset as needed can be complex and expensive.

If a distributed system does not tolerate any errors the fault of a single node can corrupt the whole system, i.e. if this node exclusively offers an essential service to the other nodes. There are several strategies to deal with faults. They will be discussed after a short classification of faults in distributed systems.

2.2.1 Classification of Faults in Distributed Systems

This section is based on [Tix09]. Another taxonomy of faults and fault-tolerance can be found in [Gär99]. Tixeuil distinguishes the *nature* of a fault, depending on whether it involves the state or the code of a node. *State-related faults* only affect – as the name says – the state of a node, i.e. the node's variables may change their values erroneously. Such errors occur e.g. due to cosmic rays or because of the continuously decreasing transistor size. *Code-related faults* compromise the node's behavior. This category includes crashes, omissions, duplications, desequencing and Byzantine faults [LSP82]. A more detailed description can be found in [Tix09].

Another criterion is the *type* of a fault. This aspect classifies the time span in which faults of arbitrary nature can occur. Three types are distinguished: *Transient faults*

are considered not to occur after a given point in the execution, i.e. there is a "last" transient error. In contrast, *permanent faults* stay permanently after a given point in the execution. *Intermittent faults* have no further limitation. Such faults can hit the system at any time. The latter type of faults is the most general one and subsumes the other two types. However, if intermittent faults do not occur too frequently, it may be sufficient to have a system tolerate transient faults provided that the time interval in which it stays operational is long enough.

A third category in the fault taxonomy of Tixeuil is the *extent* (or *span*) of the faults, describing how many components of the network can get hit by an error. In this thesis the extent of faults is insignificant.

2.2.2 Fault Tolerance and Self-Stabilizing Algorithms

Depending on the application area of the distributed system there are several approaches to deal with faults of nodes. It may be necessary that the functionality is kept up permanently. In this case, a *masking* approach is required. This category of fault tolerance hides all errors from the application, the system stays operational without restrictions. In case the continuous effective operation of the system is too expensive to guarantee or not essential a *non-masking* solution is possible: Such an approach accepts that the system does not work properly for a given time span, it suffices that it will resume its normal behavior when the fault is resolved. These two strategies lead to two major categories of fault tolerant algorithms [Tix09]:

1. *Robust algorithms* have a redundant layout for all critical components or calculations based on the expected error rate. Hence, if the system is hit by a bounded number of faults, the spare components keep the system running. Usually, robust algorithms follow a masking strategy. However, apart from being more expensive than non-masking approaches due to the additional resources for redundancy, robust algorithms require a clear concept of the (number of) errors that may occur. For instance, an algorithm that uses triple modular redundancy [vN56] can only cover up an error on a single component and may not work if another module fails.

2. *Self-stabilizing algorithms* follow a non-masking error strategy and assume all errors to be transient (cf. Section 2.2.1). Hence, no assumptions about their

nature or extent have to be made. An algorithm is self-stabilizing if it can start in any possible configuration, reaches a legitimate configuration in a finite number of steps by itself without any external intervention, and remains in a legitimate configuration [Dij74, Dol00]. Note that being able to start from any configuration implies that a self-stabilizing algorithm cannot rely on explicit initialization of variables.

The self-stabilization approach was presented by Dijkstra in [Dij74]. It did not attract much attention at first but became more and more popular in the late 1980s and has registered an increase in research activity recently [Dol00]. Some important results for classical graph problems are listed in Section 2.4.

The following definition allows to precede the formal introduction to self-stabilization with a real-world example: According to Arora and Gouda, an algorithm is self-stabilizing if the following two properties hold [AG93]:

- Convergence property: After a finite number of moves the system is in a legitimate configuration irrespective of the configuration the algorithm starts with if no further transient error occurs.
- Closure property: If the system is in a legitimate configuration, this property is preserved if no further transient error occurs.

Figure 2.1 demonstrates these properties using a well-known example. A wobbly man fulfills the convergence property since it always returns to its balanced position irrespective of its initial displacement. Having reached its stable state it will not start leaving this position by itself, hence the closure property also holds.

Note that a self-stabilizing algorithm may not be able to establish a legitimate configuration at all if faults occur too frequently, i.e. if the next error occurs before the algorithm has stabilized. Gärtner states that self-stabilizing algorithms can also deal with certain classes of permanent faults, e.g. when there is a sufficiently long error-free period of time [Gär98]. In principle this complies with the assumptions made in most publications about self-stabilization which consider all errors to be transient, i.e. no further error occurs during the stabilization process.

In the literature, two types of self-stabilizing algorithms can be found: *Silent* (or *static*) self-stabilizing algorithms stop when they have reached a legitimate configuration, i.e. no node will change its state with respect to this algorithm until the next

Figure 2.1: A real-world example for self-stabilization: A wobbly man (drawing by Christian Renner) always returns to its balanced position in finite time without external intervention, if no further impulse hits it.

fault occurs. Hence, the wobbly man (Figure 2.1) also serves as an example for a silent algorithm. Most algorithms that establish a structure on the graph, such as e.g. a matching, are silent. All self-stabilizing algorithms presented in this thesis are silent.

A *reactive* (or *dynamic*) algorithm does not terminate at all. However, it is guaranteed that once a legitimate configuration is reached, the set of legitimate configurations cannot be left. A common example for a reactive self-stabilizing algorithm is mutual exclusion [Dij74, DGT04].

2.2.3 Terms and Definitions

This section introduces the technical terminology of the area of self-stabilizing algorithms. A formal model of these terms is required by some of the proofs in this thesis. To establish a balance between mathematical symbols and readability, all terms are illustrated with the help of an intuitive self-stabilizing algorithm.

In a distributed system the communication relation is represented by an undirected graph $G = (V, E)$, with $n = |V|$ and $m = |E|$, where each process is represented by a node in V and two processes v_i and v_j are adjacent if and only if $\langle v_i, v_j \rangle \in E$. The set of neighbors of a node $v \in V$ is denoted by $N(v)$. The closed neighborhood of a node v is denoted by $N[v] = \{v\} \cup N(v)$. The diameter of G is denoted by $\mathcal{D}$ and the maximum degree of G is denoted by Δ.

In [Tur07] Turau presented a self-stabilizing algorithm for the calculation of a maximal independent set of a graph. It is shown in Algorithm 2.1. A subset $S \subseteq V$ forms

an *independent set* if no two nodes of S are adjacent. S is a *maximal independent set* if $S \cup \{v\}$ is not independent for any $v \in V \backslash S$. Figure 2.5 on page 27 shows a maximal independent set. Detailed information on such sets is provided in Section 2.4.1). The technical terms will now be explained one by one.

Algorithm 2.1 Self-Stabilizing Maximal Independent Set

Predicates:
$inNeighbor(v) \equiv \exists w \in N(v) : w.status = \mathsf{IN}$
$waitNeighborWithLowerId(v) \equiv \exists w \in N(v) : w.status = \mathsf{WAIT} \wedge w.id < v.id$
$inNeighborWithLowerId(v) \equiv \exists w \in N(v) : w.status = \mathsf{IN} \wedge w.id < v.id$

Functions:
–

Actions:
$\mathsf{R}_1 :: [status = \mathsf{OUT} \wedge \neg inNeighbor(v)]$
$\longrightarrow status := \mathsf{WAIT}$

$\mathsf{R}_2 :: [status = \mathsf{WAIT} \wedge inNeighbor(v)]$
$\longrightarrow status := \mathsf{OUT}$

$\mathsf{R}_3 :: [status = \mathsf{WAIT} \wedge \neg inNeighbor(v) \wedge \neg waitNeighborWithLowerId(v)]$
$\longrightarrow status := \mathsf{IN}$

$\mathsf{R}_4 :: [status = \mathsf{IN} \wedge inNeighbor(v)]$
$\longrightarrow status := \mathsf{OUT}$

Definition 1 (State)**.** *All nodes $v \in V$ maintain a set $\{var_1, var_2, \ldots, var_k\}_v$ of variables, each of them ranging over a fixed domain of values. The* state s_v *of the node is represented by the values of its variables.*

In the example above, the state of a node consists of a single variable *status*. The values lie in the range of IN, WAIT and OUT. In Figure 2.2 these values correspond to the colors black, gray and white. The values IN and OUT indicate whether a node is part of the maximal independent set or not, WAIT is an intermediate value that indicates that a node wants to change its *status* to IN. When Algorithm 2.1 has terminated, all nodes have their *status* variable set to either IN or OUT. If no ambiguity arises, the assignment of a value to a variable is sometimes written as an

assignment to the node, i.e. in Figure 2.2 node v_0 has the value WAIT. The states of all nodes in V represent the state of the distributed system, also called *configuration*.

Definition 2 (Configuration)**.** *A* configuration *c of the graph G is defined as the n-tuple of all nodes' states: $c = (s_{v_1}, \ldots, s_{v_n})$. The set of all configurations in G is denoted by C^G.*

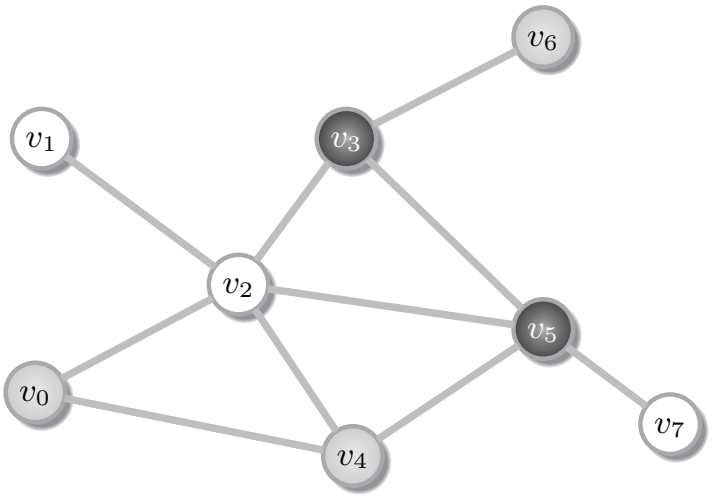

Figure 2.2: Configuration of a graph during the execution of Algorithm 2.1. The colors black, gray and white correspond to the values IN, WAIT and OUT, respectively.

Figure 2.2 shows a configuration of a graph during the execution of Algorithm 2.1. The nodes v_1, v_2 and v_7 have the value OUT assigned to their *status* variable, v_3 and v_5 (resp. the other nodes) have the value IN (resp. WAIT).

The absence of faults can be defined by a predicate $\mathcal{P}$ over the configuration. This motivates the following definition:

Definition 3 (legitimate)**.** *A configuration c is called* legitimate *with respect to $\mathcal{P}$ if c satisfies $\mathcal{P}$. Hence, a legitimate configuration is free of faults. Let $\mathcal{L}_\mathcal{P} \subseteq C^G$ be the set of all legitimate configurations with respect to a predicate $\mathcal{P}$.*

In this case $\mathcal{P}$ must evaluate to *true* if and only if the specified configuration forms a maximal independent set, i.e. for the configuration shown in Figure 2.2 $\mathcal{P}$ is *false* whereas $\mathcal{P}$ is *true* for the configuration depicted in Figure 2.5 (page 27). $\mathcal{L}_\mathcal{P}$ contains all configurations that form an independent set of the graph.

Rules specify the behavior of the nodes. Note that a node can only update its own state.

Definition 4 (Rule)**.** *A* rule *(or* action*) consists of a* name*, a* precondition *(or* guard*) and a* statement*. The precondition of a rule is a Boolean predicate defined on the state of the node itself and its neighbors' states. It decides whether a node is allowed to execute the corresponding statement. The statement describes how a node updates its state.*

The notation of a rule is:

$$\mathsf{Name} :: [\mathit{precondition}] \longrightarrow \mathit{statement}$$

Algorithm 2.1 contains four rules that define in which situations a node has to change the value of its *status* variable.

Definition 5 (Algorithm)**.** *An* algorithm *is a set of rules. It constitutes the program executed on the nodes of the distributed system.*

Definition 6 (enabled)**.** *A rule is called* enabled *in a configuration* c *if its precondition evaluates to* true *in* c*. A node is enabled in a configuration if at least one of its rules is enabled. A rule (resp. node) that is not enabled is called* disabled*.*

If several rules are enabled for a node in a configuration, one rule is nondeterministically chosen for execution. However, algorithms can be designed to guarantee that at most one rule is enabled per node for any configuration. This can be done by extending the guards of the rules to include the negation of the other rules' guards. Hence, without loss of generalization it is assumed that a node is enabled for at most one rule in a given configuration.

In the configuration depicted in Figure 2.2 all nodes are enabled, except for nodes v_2 and v_7. They are disabled since they have a neighbor (e.g. v_5) that is included in the minimal independent set and they themselves are not. Nodes v_4 and v_6 are enabled to execute rule R_2 to set their *status* variable to OUT. The black nodes are neighbors, and hence, both of them are enabled to leave the independent set (rule R_4). Node v_0 could set its *status* to IN via rule R_3 and node v_1 is enabled to execute rule R_1 to set its *status* to WAIT. The execution of a rule by a node is called a *move*.

Definition 7 (Move)**.** *A* move *is a tuple* $(s, s')_v$*, where* s *(resp.* s'*) denotes the state of node* v *before (resp. after) the execution of the statement of an enabled rule.*

If it is clear (or of no relevance) which node executes the move, the subscript will be omitted. If a certain rule is enabled for a given node, the corresponding move is called *enabled* also. An essential property of the system is its *synchrony*. In Figure 2.2 the nodes v_3 and v_5 are both enabled to execute rule R_4. If they make a move simultaneously, both of them set their *status* variable to OUT since they read their neighbors' states at the same time. However, if one of them makes its move first, the other node becomes disabled since it no longer has a black neighbor. The synchrony of a distributed system is modeled by a *scheduler* (or *daemon*). For a given configuration the scheduler chooses which nodes make a move simultaneously.

Definition 8 (Scheduler)**.** *The* scheduler *of a distributed system is a function* $sched : C^G \hookrightarrow 2^V$*, such that* $sched(c)$ *is a nonempty subset of the nodes in* V *that are enabled in configuration* c*.*

The most common schedulers are:

- the *central* scheduler: At any time, only *a single node* makes its move, i.e $\forall c \in C^G : |sched(c)| = 1$.
- the *synchronous* scheduler: *All* enabled nodes make their moves simultaneously.
- the *distributed* scheduler: *Any nonempty subset* of the enabled nodes can make their moves simultaneously.

Although it is easier to prove stabilization for algorithms working under the central scheduler, the synchronous and the distributed scheduler are more suitable for practical implementations. The distributed scheduler allows the nodes to operate with different speed, i.e. not all nodes have to make their move at the same time. Note that the distributed scheduler subsumes the other two types of schedulers and is the most general concept. In general, schedulers have no restrictions on their scheduling policy. However, sometimes it is useful to assume *fairness*:

Definition 9 (Fairness)**.** *A scheduler is called* fair *if it prevents a node being continuously enabled without making a move. Otherwise, the scheduler is called* unfair.

The results presented in this thesis are valid for the unfair distributed scheduler if not explicitly stated otherwise.

Self-stabilizing algorithms operate in *steps*. Intuitively, steps can be seen as time intervals, such that every node can make at most one move within one step and such that all nodes make their move simultaneously. This implies that for any step all nodes read their neighbors' states at the same time.

Definition 10 (Step)**.** *A* step *is a tuple* (c, c')*, where* c, c' *are configurations, such that*

- *all nodes that make a move in this step are enabled in configuration* c*, and*
- c' *is the configuration reached after these nodes have made their move* simultaneously.

When the central scheduler is used, each step consists of the move of a single node only. Thus, if a step consists of the move $\mathsf{m} = (s, s')$ that transforms configuration c_0 into c_1 it is also possible to write $\mathsf{m} = (c_0, c_1)$ and with a slight abuse of notation $\mathsf{m}(c_0) = c_1$. This notation does not introduce any ambiguity when the central scheduler is used, since c_0 and c_1 coincide in all components but one.

Definition 11 (Execution)**.** *An* execution *of an algorithm is a maximal sequence* $c_0, c_1, \ldots$ *of configurations such that for each configuration* c_i *the next configuration* c_{i+1} *is obtained from* c_i *by a single step.*

With these terms and definitions it is possible to describe the two properties *closure* and *convergence* (cf. Section 2.2.2) more formally, which are used to give a formal definition of self-stabilization:

Definition 12. *An algorithm is* self-stabilizing *with respect to* $\mathcal{P}$ *if the following two properties hold:*

- *Closure property: For all configurations* $c_0, c_1 \in C^G$*: If* (c_0, c_1) *is a step with* $c_0 \in \mathcal{L}_\mathcal{P}$*, then* $c_1 \in \mathcal{L}_\mathcal{P}$*.*
- *Convergence property: For every execution* $c_0, c_1, \ldots$ *there is an integer* i *such that* $c_i \in \mathcal{L}_\mathcal{P}$*.*

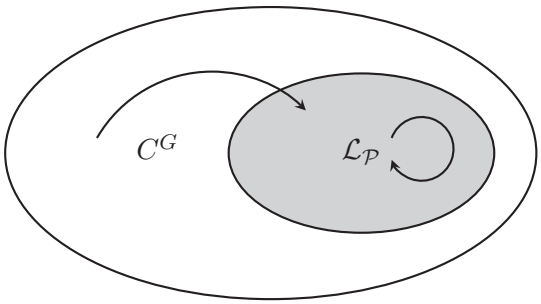

Figure 2.3: Closure and convergence

Definition 12 is illustrated in Figure 2.3: The set $\mathcal{L}_\mathcal{P}$ of legitimate configurations is a subset of C^G, the set of all configurations. Any step starting from a legitimate configuration results in another legitimate configuration. If the initial configuration is not in $\mathcal{L}_\mathcal{P}$, then in a finite number of steps a legitimate configuration is reached.

More details and other elaborative introductions to self-stabilization can be found e.g. in [Dol00], [Tel01], or [Tix09].

2.2.4 Complexity of Self-Stabilizing Algorithms

The *complexity* of an algorithm is a measure for its maximum resource demand. Usually this demand depends on the size of the input or, in case of a distributed algorithm, the number of processors. The considered resources can be time, memory, or the number of messages sent. The latter does not apply in this thesis due to the use of the state model (see Section 2.1) [AW04]. Garey and Johnson contributed the most influential publication on complexity of problems and algorithms [GJ79]. However, they focus on centralized algorithms. The complexity of distributed algorithms with respect to the communication model is discussed e.g. in [AW04]. A detailed introduction to the complexity of self-stabilizing algorithms can be found in [Dol00].

There are several measures for the time complexity of a self-stabilizing algorithm. Note that these measures do not consider local computation of the nodes. This is due to the assumption that the time needed for communication greatly exceeds the time needed for computation, an assumption made for algorithms that consider the computations to be based on local knowledge only. A detailed discussion on this topic can be found in [Tel01]. A standard measure is the *move complexity*.

Definition 13 (Move Complexity)**.** *The (worst-case)* move complexity *of a self-stabilizing algorithm denotes the maximum number of individual moves needed to reach a legitimate configuration irrespective of the initial configuration.*

This upper bound is relevant for many practical applications such as wireless systems with bounded resources. The execution of self-stabilizing algorithms defined for the state model in a wireless setting requires a transformation. The cached sensornet transform (CST) proposed by Herman is a widely used transformation technique [Her04]. It requires that nodes broadcast their state to their neighbors after every move. Since communication is the main consumer of energy, a reduction of the number of broadcasts prolongs the lifetime of a network [TW09].

For the second standard measure for time-complexity of a self-stabilizing algorithm, assume the synchronous scheduler. In this case, in any step *all* enabled nodes make a move. The term *(asynchronous) rounds* tries to extend this idea to match the nature of the central and the distributed scheduler [Dol00]. Starting from a given configuration some nodes may be scheduled several times before all enabled nodes have made a move. Furthermore, since the move of a node can disable other nodes, it does not make sense to require all nodes that were enabled at the beginning of a round to make a move until the round is completed. It also suffices when a node is disabled in between. Note that only for the synchronous scheduler the number of moves per round is limited to the number of nodes, since a round is a single step under this scheduler.

Definition 14 (Round)**.** *A round is a minimal sequence of steps during which any node that was enabled at the beginning of the round has either made a move or has become disabled at least once.*

Definition 15 (Round Complexity)**.** *The (worst-case)* round complexity *of a self-stabilizing algorithm denotes the maximum number of rounds needed to reach a legitimate configuration irrespective of the initial configuration.*

Considering rounds allows to make assumptions on the states of all nodes, e.g. after the first round all nodes have assigned certain values to their variables. The round complexity further permits to ignore scenarios in which a particular node is continuously enabled but does not make a move. The current round does not end unless the node either makes a move or the move of one of its neighbors disables it.

The worst-case number of moves or rounds does not necessarily reflect the time the algorithm needs to stabilize. The number of moves alone does not provide the information whether these moves are equally distributed among all nodes or whether they are performed by a small group of nodes only. Hence, only for the central scheduler, this number conforms exactly with the worst-case stabilization time. On the other hand, a round has no fixed limit for the number of moves contained under the central or the distributed scheduler. Counting the worst-case number of *steps* estimates the time an algorithm needs to stabilize best.

Definition 16 (Step Complexity)**.** *The (worst-case)* step complexity *of a self-stabilizing algorithm denotes the maximum number of steps needed to reach a legitimate configuration irrespective of the initial configuration.*

Note that for the central scheduler the step complexity is equivalent to the move complexity, since this scheduler allows only one move per step. For the synchronous scheduler the step complexity is equivalent to the round complexity, since under this scheduler a round consists of exactly one step. For the distributed scheduler the time a self-stabilizing algorithm needs to reach a legitimate configuration exactly corresponds to the number of steps in the execution. However, since any execution under the central scheduler is also valid for the distributed scheduler, its worst-case number of steps cannot be smaller than the move complexity under the central scheduler. Usually, the step complexity is merely used for the synchronous scheduler to emphasize that the rounds are synchronous.

The last complexity measure considered in this thesis refers to the memory requirement of an algorithm. Often, self-stabilizing algorithms run on very restricted hardware, therefore it is important to use the resources economically.

2.3 Design Methods for Self-Stabilizing Algorithms

The definition of a legitimate configuration for a given problem is usually described by several individual properties that have to hold true. In general, a self-stabilizing algorithm consists of a set of rules that perform a local check whether a precondition of a rule is valid for the executing node and set the state accordingly, if necessary.