1 Introduction

With the increasing communication and networking capabilities of computers, a new field of research emerged within computer science: Distributed Computing. Its focus is the design and study of distributed algorithms, which are algorithms especially designed to run in distributed systems. The first conference on this subject took place in the year 1982 [POD82]. A distributed system consists of a set of nodes, also called processes or processors, connected by communication links. Examples of distributed systems are several computers connected via a network like the Internet and several microprocessors connected via wireless technology such as wireless sensor networks. The size of distributed systems can range from tens or hundreds to millions of nodes.

A distributed algorithm determines the behavior of each node in the distributed system. There is no central unit of control. Instead, the nodes are autonomous. The goal of distributed algorithms is to establish a certain global behavior of the distributed system as a whole. However, in order to achieve a particular global behavior, the distributed algorithm has to cope with two obstacles: locality and non-determinism. Locality refers to the fact that the view of a node is limited to a small part of the distributed system. Nodes have to cooperate in order to obtain information outside the local view of a node or regarding the structure of the distributed system as a whole.

Non-determinism is inherent to distributed systems. Messages may arrive in an order different from the one that they have been sent in. Distributed systems are not necessarily homogeneous. As an example, consider the Internet which is quite heterogeneous regarding speed and latency of communication links as well as the computational power of the nodes. Communication links range from only a few kilobits to multiple gigabits per second. Nodes range from small embedded systems and smart phones to fast server systems. The exact characteristics of links and nodes are typically unknown in advance. Hence, distributed algorithms are designed to be ignorant about them so that they function in any setting.

The size of distributed systems is constantly growing. Again, the Internet is a good example of that, since more and more devices are constantly

connected to it, e.g., smartphones. However, with the rising number of nodes and communication links, the probability of faults increases [KP93]. To cope with faults, fault-tolerant distributed algorithms were introduced.

Informally, a distributed algorithm is fault-tolerant if it is able to recover from faults in finite time or if it is able to provide some functionality in spite of faults. Note that the class of tolerated faults may be restricted, i.e., a distributed algorithm tolerates faults of a certain kind but may become dysfunctional after faults of a different kind. Human interaction might be necessary, possibly involving a manual reset of the distributed system in order for the distributed algorithm to recover. Dijkstra introduces a class of distributed algorithms that provide a large degree of non-masking fault-tolerance: self-stabilizing distributed algorithms. Given a large enough window of execution time without faults, a self-stabilizing distributed algorithm can recover from any transient fault that might have happened in the past without human intervention, regardless of the fault's scale or nature. Lamport called the work of Dijkstra a "milestone in work on fault-tolerance" [Lam84].

Many techniques have been proposed to build fault-tolerant distributed algorithms. It can be observed that any fault-tolerant distributed algorithm implements two functions: detection of faults and corrections of faults. However, distributed systems are a particularly difficult setting to detect and correct faults. The reasons for that are related to both locality and non-determinism. Locality makes it difficult to detect faults. The main problem is that something might look correct locally but is incorrect from a global perspective. As an example, we refer to spanning tree construction. A fault may change the parent of a node within the tree in such a way that a loop is created. The loop may span across several hundred nodes and communication links. Keeping additional information on each node often facilitates making solutions locally checkable. Informally, this means that at least one node will eventually become aware of faults, using only the information within the local view of the node. The additional information is usually redundant from the point of view of a perfect distributed system. As an example, consider a spanning-tree protocol that, per node, stores a pointer to the parent within the tree and the distance to the root of the tree. The distances are implied by the pointers. Nonetheless, making nodes aware of their distance to the root node plays an important role in detecting faults in self-stabilizing spanning tree protocols.

The non-determinism accelerates a process called contamination. Even before a process has had a chance to detect and correct a fault, faulty information may have been passed on to other nodes. Due to locality, the

data is not recognized as faulty as it is passed on to further nodes. Faulty information can very easily spread to large parts of the network. These parts of the network may temporarily stop exhibiting the desired behavior. The contamination process is hard to reverse.

Several techniques have been considered to design fault-tolerant algorithms. However, many of them involve performing a global reset, after a fault has been detected. The distributed algorithm will then start from the beginning. This technique was also proposed to make distributed algorithms self-stabilizing. However, with the increasing size of distributed systems, global resets becomes more and more undesirable as they temporarily affect the entire system.

This has been the motivation for finding ways to deal with faults more locally. One essential ingredient is to spatially bound the contamination process, i.e., containing the effects of a fault to a small area around the location of the fault. The second ingredient is to locally repair the fault. Kutten and Peleg proposed a technique called "fault-local mending". A similar approach is the notion of fault-containment, which is an extension of self-stabilization. Fault-containing self-stabilizing distributed algorithms not only recover from transient faults of any scale or nature, but also prevent contamination and guarantee recovery from small-scale faults within constant time. The repair of small-scale faults happens locally with minimal effect on other parts of the network.

Two important properties of fault-containing self-stabilizing algorithms are the containment time and the fault-gap. The former describes how fast a small-scale fault can be repaired. The latter describes how soon another fault can be contained, i.e., how long it takes the algorithm to be prepared for another fault. Thus, having a low fault-gap is essential if small-scale faults are expected to happen frequently. Fault-containment has already been the subject of another Ph.D. thesis [Gup97]. Besides presenting several problem specific fault-containing self-stabilizing algorithms, that thesis presents a general technique to augment any silent self-stabilizing protocol with fault-containment via an automatic transformation. The transformation utilizes global synchronization and global reset to achieve fault-containment. While the containment time is constant, the fault-gap of the proposed transformation is linear in the number of nodes in the system. Furthermore, the number of nodes is bounded by a constant which has to be known before the algorithm is deployed. As a result, the proposed containment algorithms do not scale to larger networks at runtime.

Note that topological faults (e.g., removal or addition of new edges) are in general hard to deal with. Some problems, like constructing a shortest path

tree, force algorithms to reconstruct large parts of the tree after a topological fault. Hence, often neither a fast recovery from topological faults nor dealing with them locally is possible. Techniques like super-stabilization therefore focus on another aspect of recovering from topological faults: while the system recovers from the topology change, it must not violate a certain safety property.

1.1 Contributions of the Thesis

This thesis presents a novel approach to fault-containment based on a technique for local synchronization. It is utilized to create an automatic transformation that adds fault-containment to any silent self-stabilizing distributed algorithm. The transformation does not only achieve a constant containment time but also a constant fault-gap. At the same time, the transformation increases the stabilization time of the original algorithm by merely a constant factor. The impact of a fault is confined to an area of small size around the fault's location. Outside this area, none of the nodes execute an action or change their variables. This is a considerable improvement over previously known transformations for fault-containment. They have a fault-gap of $\Omega(n)$ and a small-scale fault is allowed to globally disrupt the variables added by the transformation.

The transformation creates backups of the local state of each node. The backups are placed on neighbors of each node. Previously known transformations create up to Δ backups. We show that the number of backups per node can be reduced to two. The second contribution of this thesis is a self-stabilizing algorithm that computes a placement for two (or more) backups per node in such a way that the standard deviation of the number of backups stored per node assumes a local minimum. It is shown how this algorithm can be incorporated into the transformation such that containment time and fault-gap remain constant.

The third main contribution consists of the introduction of the concept of fault-containing super-stabilization. It describes distributed algorithms which withstand small-scale state corruptions and topology changes, even if they happen at the same time. An automatic transformation is presented, with which any silent super-stabilizing algorithm can be made fault-containing super-stabilizing. The fault-containing super-stabilizing algorithm will first correct the state corruption and then exhibit the super-stabilizing behavior of the original protocol. The safety property of the original super-stabilizing protocol holds after a constant number of rounds.

The number of backups is reduced to a minimum of at most four backups per node. The backups are placed within the two-hop neighborhood of each node.

All of the above protocols are shown to work under the most general system model: the unfair distributed scheduler. In order to prove self-stabilization under this model, a novel technique is introduced: serialization. It can be used to elevate proofs written under the assumption of the central scheduler, a rather strong system model, to the more general distributed scheduler.

1.2 Organization of the Thesis

Chapter 2 defines the formal model of distributed systems used in this thesis. Chapter 3 gives an overview of fault-tolerance in distributed systems, including an introduction to the notions of masking and non-masking fault-tolerance and self-stabilization.

Chapter 4 discusses various methods for proving that a given distributed algorithm is self-stabilizing and introduces the novel method of serialization. In addition, composition methods are discussed which are used in the construction of the algorithms presented in this thesis. Chapter 5 presents the transformation which adds fault-containment to any silent self-stabilizing algorithm. Chapter 6 presents a self-stabilizing algorithm that computes a backup placement such that the standard deviation of the number of backups that each node stores assumes a local minimum. Furthermore, it is shown how this placement algorithm can be integrated into the transformation presented in Chapter 5 without increasing the containment time or fault-gap. Chapter 7 introduces the concept of fault-containing super-stabilization and presents a transformation that converts any silent super-stabilizing algorithm into a fault-containing super-stabilizing algorithm.

2 Preliminaries

This chapter describes the formal models of distributed systems, algorithms, and their execution which are used in this thesis. They are commonly used in research on self-stabilizing and fault-containing algorithms.

2.1 Distributed Systems

A distributed system consists of *nodes*, also sometimes called processors or processes, which are connected by communication links. The *topology* of a distributed system is modeled as an undirected graph $G = (V, E)$ where V denotes the set of nodes and each edge $(v_1, v_2) \in E \subseteq V \times V$ corresponds to a communication link between nodes v_1 and v_2. Two nodes connected by an edge (i.e., a communication link) are called *neighbors*. For a node $v \in V$, the set $N(v)$ denotes the open neighborhood of v, i.e., $N(v)$ contains all neighbors of v but not v itself. The closed neighborhood of v is denoted by $N[v] = N(v) \cup \{v\}$. Furthermore let $\deg(v) = |N(v)|$ denote the degree of node v and $\Delta = \max\{\deg(v) \mid v \in V\}$ the maximum degree of the nodes. By $n = |V|$ we denote the number of nodes, by $m = |E|$ the number of edges in the system, and by D the diameter of the topology G. The diameter is defined as the longest shortest path between any pair of nodes. Examples of such distributed systems are all computer or sensor networks.

It is not assumed that nodes have access to clocks. Hence, nodes cannot measure the time that has passed. Furthermore, there is no central unit of control, meaning that there is no entity coordinating the action of the nodes or the communication between them. Control is distributed among the nodes. Also, there is no global view. Thus nodes only have access to the data stored in their own memory and any data that is obtained by communicating with neighbors.

2.2 Algorithms, Protocols, and State Model

A *distributed algorithm* $\mathcal{A}$ is a mapping which assigns a finite set of protocols to each node $v \in V$. Informally, $\mathcal{A}(v)$ denotes the set of protocols that node v executes. The pair $(v, P), v \in V, P \in \mathcal{A}(v)$ is called an *instance* of protocol

P on node v. For protocols, the notation of repetitive constructs as defined by Dijkstra is used [Dij75]. A *protocol* consists of a set of *rules* separated by $\square$ and enclosed by the keywords **do** and **od**. Each rule is a guarded command of the form

$$guard \longrightarrow statement; statement; \ldots$$

Associated with each protocol $P \in \mathcal{A}(v)$ with $v \in V$ is a finite set of variables. Each variable has a name and a domain. For any given $v \in V$, the names of the variables of any pair of protocol in $\mathcal{A}(v)$ are assumed to be disjoint. The variable with the name "x" of a protocol $P \in \mathcal{A}(v)$ is denoted by $v.x$. The values of these variables constitute the *local state* of instance (v, P). The tuple[1] of the local states of all instances $(v, P), P \in \mathcal{A}(v)$ constitutes the *local state* of node v. The *guard* of a rule is a Boolean predicate. The guards and statements of (v, P) may read only the variables of all instances (u, Q), $u \in N[v]$, $Q \in \mathcal{A}(u)$, i.e., all instances within the closed neighborhood of v. The statements of (v, P) may only modify the variables of (v, P). This implies that access to variables of instances outside the closed neighborhood is not allowed. This communication model is called the *locally shared memory* model.

The tuple of the local states of all nodes $v \in V$ constitutes the *configuration* of the distributed system, often also called the global state. The set of all possible local states of an instance (v, P) is denoted by σ_P and the set of all possible local states of a node by $\sigma_{\mathcal{A}}$. $\Sigma_{\mathcal{A}}$ denotes the set of all possible configurations. $\mathcal{I}_{\mathcal{A}} = \{(v, P) \mid v \in V \wedge P \in \mathcal{A}(v)\}$ denotes the set of all instances. Two instances $(v, P) \neq (v', P')$ are said to be neighboring if $v' \in N[v]$. In particular, the two instances on the same node (i.e., $v = v'$) are called neighboring.

A rule of an instance (v, P) is called *enabled* if its guard evaluates to true. An instance (v, P) is called *enabled* if at least one of its rules is enabled. Node v is called *enabled* if at least one instances $(v, P), P \in \mathcal{A}(v)$ is enabled. It is said that an Algorithm $\mathcal{A}$ has terminated in configuration c if all instances $(v, P) \in \mathcal{I}_{\mathcal{A}}$ are disabled in c. Note that this model permits multiple protocols per node. Informally, it can be said that these protocols are executed in parallel. This is formalized in Section 2.4.

Figure 2.1 shows a simple example: the protocol *MIS*. The corresponding algorithm $\mathcal{A}_{MIS}$ satisfies $\mathcal{A}_{MIS}(v) = \{MIS\}$ for all $v \in V$, i.e., an instance of protocol *MIS* exists on every node. This algorithm will serve as an example throughout this thesis. It will become clear that under certain assumptions,

[1] of fixed but arbitrary order

Protocol MIS

Nodes: v is the current node
Variables: $v.s \in \{\text{Out}, \text{In}\}$
Predicates: $inNeighbor(v) \equiv \exists u \in N(v) : u.s = \text{In}$

do
[M1] $v.s = \text{Out} \wedge \neg inNeighbor(v) \longrightarrow v.s := \text{In}$
[M2] $\square\ v.s = \text{In} \wedge inNeighbor(v) \longrightarrow v.s := \text{Out}$
od

Figure 2.1: A protocol for computing a maximal independent set [SRR95, HHJS03]

the algorithm computes a maximal independent set consisting of all nodes $v \in V$ with $v.s = \text{In}$.

In large part, the topology is assumed to be fixed in this thesis. An exception is Chapter 7, which considers the case where the topology may change as a result of faults. In this case, the set of edges is regarded as part of the systems configuration. We define the notions of extended configuration and extended execution. An *extended configuration* consists of the pair (E, c) where $E \subseteq V \times V$ denotes a set of edges and $c \in \Sigma_{\mathcal{A}}$ is a configuration. The set of all possible extended configurations is denoted by $\Sigma_{\mathcal{A}}^{ext}$.

To conveniently refer to the value of an expression in a certain configuration, the notation $c \vdash \textit{expression}$ is introduced. For example, $c \vdash v.s$ refers to the value of the variable $v.s$ in a configuration $c \in \Sigma_{\mathcal{A}}$ and $c \vdash v.s = \text{In}$ is true if and only if the variable $v.s$ has the value In in c. Furthermore, writing $(c \vdash v.s) = \text{In}$ is equivalent to $c \vdash (v.s = \text{In})$ and thus, the parentheses are omitted. The notation may also be combined with extended configuration, e.g., $(E, c) \vdash N(v) = \emptyset$ is true if and only if node $v \in V$ has no neighbors in E. The notation $c|_m$ is used to refer to the local state of an instance $m \in \mathcal{I}_{\mathcal{A}}$ in configuration c.

2.3 Execution Model and Schedulers

For different distributed systems, the speeds and latencies of the underlying communication links and the computational power of the nodes may vary significantly. But also for one distributed system, these parameters may vary over time. Hence, the exact parameters are usually unknown in ad-

vance. This inherent non-determinism is modeled as part of the execution model via a virtual entity called the scheduler. Several scheduler models exist. Three scheduler models are used in this thesis. The distributed scheduler is the most general one. Furthermore, two special cases of the distributed scheduler exist: the central and the synchronous scheduler.

Execution of a distributed algorithm $\mathcal{A}$ is organized into *steps*. Let $c_{i-1} \in \Sigma_{\mathcal{A}}$ denote the configuration of the distributed system before the i-th step. The scheduler selects a non-empty subset $S_i \subseteq \mathcal{I}_{\mathcal{A}}$ of instances that are enabled in c_{i-1}. Next, all selected instances $(v, P) \in S_i$ make a *move*, i.e., the statements of one enabled rule are executed. Note that several models exists for the case where multiple rules of a single instance are enabled. In this thesis, this case is avoided, i.e., the algorithms are constructed in such a way that at most one rule is enabled at a time. Furthermore, S_i may contain two or more neighboring instances. Clarification is needed on how a move of an instance (v, P) affects moves of neighboring instances during the same step. *Composite atomicity* is assumed, which means that choosing an enabled rule and executing its statements is regarded as one atomic block. The changes made by an instance (v, P) become visible to neighboring instances at the end of the i-th step, i.e., after all neighboring instances have made their move. This also holds for two neighboring instances on the same node. When all instances in S_i have made their move, this yields c_i, the configuration after the i-th step and before the $(i+1)$-th step of the execution.

The *distributed scheduler* is not restricted in its choice of S_i and it is assumed to choose S_i non-deterministically [BGM89]. That means, no model (e.g., probabilistic or deterministic) for predicting the choice of S_i exists. Besides having to choose a non-empty S_i, the distributed scheduler can select any number of enabled instances in each step. The *synchronous scheduler* chooses all instances enabled in c_{i-1} [Her90]. It is the only deterministic scheduler, and assuming that all protocols are deterministic, c_i is uniquely determined by c_{i-1}. It models synchronous distributed systems for which it is true that any changes to the local state of node v can be propagated to the neighboring nodes in constant time. Furthermore, all nodes work in synchrony in the sense that they execute their moves simultaneously at any time. The distributed scheduler on the other hand models asynchronous systems, in which no such assumptions exist. It takes into account that in heterogeneous systems the speed of nodes and links may vary. Hence, in fast areas more moves may be made than in slow areas. Note that the synchronous scheduler is a special case of the distributed scheduler. Hence,

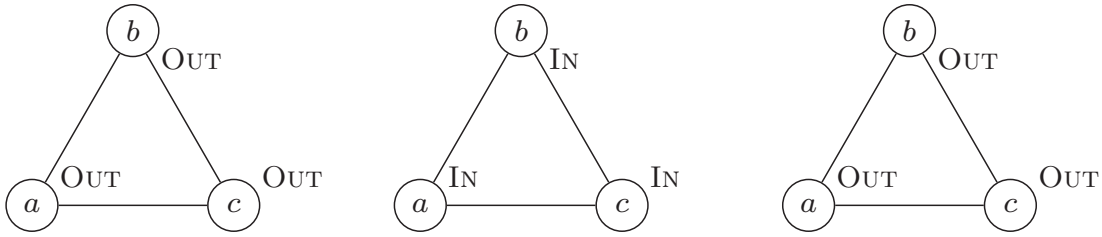

Figure 2.2: Execution of the MIS algorithm in a synchronous system

any algorithm designed for the distributed scheduler also works under the synchronous scheduler.

The *central scheduler* is the third type. It selects only one instance per step, i.e., $|S_i| = 1$ for all i. This scheduler simplifies the design of distributed algorithms as it provides mutual exclusion between neighboring instances. Furthermore, the mutual exclusion provides some form of symmetry breaking. For algorithms that do not perform symmetry breaking explicitly, executions under the distributed scheduler may lead to a livelock. Figure 2.2 shows an example of such a livelock, namely an execution of the protocol shown in Figure 2.1 under the synchronous scheduler.

The central scheduler has a long tradition in self-stabilizing research as the first self-stabilizing algorithm was designed for this particular scheduler [Dij74]. Like the synchronous scheduler, the central scheduler is a special case of the distributed scheduler. However, the central scheduler is asynchronous.

Schedulers may be further categorized by the level of fairness they provide [DTY08]. A scheduler is called *weakly fair* if it eventually selects any continuously enabled instance. Stronger types of fairness are discussed in [DTY08]. The most general scheduler is the *unfair* scheduler which does not provide any type of fairness. For example, it may never select a continuously enabled instance, provided that there are other enabled instances. The synchronous scheduler is obviously weakly fair. Distributed and central schedulers may exhibit any of the three types of fairness. All algorithms presented in this thesis work under unfair schedulers.

The sequence $e = \langle c_0, c_1, c_2, \ldots \rangle$, where c_0 denotes the initial configuration and $c_i \in \Sigma_{\mathcal{A}}$ is the configuration after the i-step, is called an *execution*. It describes the behavior of a distributed system over time. The corresponding sequence $S = \langle S_1, S_2, S_3, \ldots \rangle$ is called the *schedule* of the execution. Each transition from c_i to c_{i+1} is atomic, i.e., intermediate configurations in-between c_i and c_{i+1} do not exist in this model. An execution ends when

the algorithm has terminated. This thesis discusses silent algorithms only. An algorithm is called *silent* if it terminates after a finite number of steps. Hence, all executions considered in this thesis are finite. Whether an algorithm terminates may depend heavily on the scheduler as discussed in this section using the example given in Figure 2.2.

Note that unlike in topological self-stabilization [GJR+10], it is not assumed that instances are able to actively destroy or create communication links. Hence, for an execution starting in an extended configuration (E, c), the set of communication links is assumed to be equal to E at all times.

In this thesis we say that a Boolean predicate p on $\Sigma_{\mathcal{A}}$ is *stable* for an execution e of $\mathcal{A}$ if any configuration in e subsequent to a configuration satisfying p also satisfies p.

The notation $(c : S)$ is used to describe the configuration after a step of an algorithm $\mathcal{A}$ under the distributed scheduler, where c denotes the configuration before the step and $S \subseteq \mathcal{I}_{\mathcal{A}}$ the set of instances enabled in c that make a move during the step. If the algorithm at hand is deterministic, then $(c : S)$ denotes the configuration after the step which is uniquely determined. $(c : S_1 : S_2 : S_3 : \ldots)$ is defined to be equivalent to $(\ldots(((c : S_1) : S_2) : S_3) : \ldots)$. Otherwise, if the algorithm is not deterministic (e.g., probabilistic or non-deterministic), $(c : S)$ denotes the set of all configurations that are possible outcomes of the step. $(c : S_1 : S_2)$ denotes the set $\bigcup_{c' \in (c:S_1)} (c' : S_2)$, $(c : S_1 : S_2 : S_3)$ denotes the set $\bigcup_{c' \in (c:S_1:S_2)} (c' : S_3)$ and so forth.

A step of $\mathcal{A}$ under the central scheduler is described by $(c : m)$ where $m \in \mathcal{I}_{\mathcal{A}}$ is an instance enabled in c. It is defined as $(c : m) = (c : \{m\})$. Again, $(c : m)$ may denote a single configuration or a set of configurations depending on whether the algorithm at hand is deterministic or not. Multiple steps under the central scheduler are denoted by $(c : m_1 : m_2 : m_3 : \ldots)$ which is equivalent to $(c : \{m_1\} : \{m_2\} : \{m_2\})$. Both notations may be combined, e.g., $(c : m_1 : S_1)$.

2.4 Notions of Time

In this section, we define three different ways to measure time that are commonly used in self-stabilizing research: move, step, and round complexity. Consider a finite execution $e = \langle c_0, c_1, \ldots, c_k \rangle$ and the corresponding schedule $S = \langle S_1, S_2, \ldots, S_k \rangle$. The number of moves in e is defined as $\sum_{i=1}^{k} |S_i|$, which is the total number of moves made by all instances. The length of e in steps is k.

The length of e in *rounds* is determined by partitioning e into rounds as follows: The first round of e is defined as the prefix $\langle c_0, c_1, \ldots, c_j \rangle$ of e with minimal length such that $V \setminus D_0 \subseteq \bigcup_{i=1}^{j} S_i \cup D_i$ where $D_i \subseteq \mathcal{I}_{\mathcal{A}}$ is the set of instances that are disabled in c_i. All further rounds are derived recursively by applying the definition of the first round to the suffix $e' = \langle c_j, c_{j+1}, \ldots, c_k \rangle$ and the corresponding schedule $S' = \langle S_{j+1}, S_{j+2}, \ldots, S_k \rangle$, i.e., the second round of e is the first round of e' and so on.

Counting the number of moves is a good measure to estimate the energy consumption in settings where transmission of data is costly. As a node has to broadcast its local state to neighbors after each state-change, reducing the number of moves may save energy [TW09]. Using the number of moves as a measure of time is justified for the central scheduler, under which at most one move is made in each step. However, in general that does not hold. The number of moves in each step that the scheduler model allows, may be seen as the degree of parallelism that the model allows. The central scheduler exhibits the lowest degree of parallelism, hence the number of moves is identical to the number of steps. The synchronous scheduler provides the maximum degree of parallelism. Hence one round is completed each step. In general it holds

$$rounds \leq steps \leq moves$$

Informally, a round can be described as the shortest prefix of an execution that allows any information to travel at least one hop. It takes at least 1 step to complete one round. An upper bound on the number of steps per rounds highly depends on the scheduler model and the algorithm at hand. However, we argue that in practice the time needed to complete one round depends on the slowest communication link and the largest latency. Hence, we assume that the time per round is bounded by some system dependent expression.

Note that under an unfair scheduler, rounds can potentially be of infinite length as the scheduler is not obligated to eventually select a continuously enabled node. However, under the weakly fair scheduler, a round is guaranteed to be finite. Furthermore, rounds are guaranteed to be finite even under an unfair scheduler if the algorithm is silent. This is the case for all algorithms in this thesis.

Generally, the time that a node needs to make a move is disregarded. It is assumed to be small compared to the time needed for communication.

The next two lemmas clarify that the definition of rounds indeed satisfies the very intuitive requirement that the length of any suffix of an execution

of x rounds does indeed not exceed x rounds. We use the operator $\circ$ to denote the concatenation of sequences.

Lemma 2.1. *Let e_1 be an execution of a distributed algorithm $\mathcal{A}$ and r_1 the first round of e_1 such that $e_1 = r_1 \circ e_1'$. Let e_2 denote a suffix of e_1 and r_2 the first round of e_2 such that $e_2 = r_2 \circ e_2'$. Then e_2' is a suffix of e_1'.*

Proof. Only the case where e_1' is a proper suffix of e_2 is considered. Otherwise, the claim is obviously true. Let r_0' be a prefix of r_1 and r_1' a suffix of r_1 such that $e_1 = r_0' \circ e_2$ and $r_1 = r_0' \circ r_1'$. The claim holds if r_1' is a prefix of r_2, which we proceed to prove.

Let $r_1 = \langle c_0, c_1, \ldots, c_k \rangle$. There exists an instance $m \in \mathcal{I}_\mathcal{A}$ that is enabled in all $c_0, c_1, \ldots, c_{k-1}$ and that is executed or becomes disabled during the transition $c_{k-1} \to c_k$, but not earlier. If c_k is the first configuration of r_2, then r_1' is clearly a prefix of r_2. Otherwise, some c_i, $i < k$ is the first configuration of r_2 and m is enabled in c_i. Hence, r_2 must include c_k and thus r_1' is a prefix of r_2. □

Lemma 2.2. *Let e be an execution of x rounds. Any suffix of e has at most x rounds.*

Proof. Let e be an execution and e' a suffix of e. Assume that e is partitioned into rounds $r_1, r_2, \ldots, r_k$ and e' into rounds $r_1', r_2', \ldots, r_l'$. Let $e_i = r_i \circ r_{i+1} \circ \ldots \circ r_k$ and $e_i' = r_i' \circ r_{i+1}' \circ \ldots \circ r_l'$. The term e_{k+1} is defined to be the empty sequence. By induction on j it is shown that any e_j' is a suffix of e_j. In particular, e_{k+1}' is a suffix of e_{k+1} which is the empty sequence. Hence, e_{k+1}' is the empty sequence and thus $l \leq k$. By assumption e_1' is a suffix of e_1. Assume that e_j' is a suffix of e_j for $j \leq k$. By Lemma 2.1 it follows that e_{j+1}' is a suffix of e_{j+1}. □

2.5 Other Communication Models

The algorithms in this thesis are designed for the locally shared memory model with composite atomicity as defined in Sections 2.2 and 2.3. This section gives an overview of other communication models that are commonly used.

Theoretical models exist, in which an instance (v, P) can read any variables within distance k of v. This is called the *distance-k model*. For $k = 1$ it matches the model presented in Section 2.2. Values $k \geq 2$ often make it easier to design distributed algorithms. However, this model is very costly to emulate in the distance-1 model [GGH$^+$04, GHJT08, Tur12].

Dolev et al. studied the effects of relaxing the assumption of composite atomicity [DIM90, DIM93]. They propose a model called read/write-atomicity. Nodes communicate via communication registers instead of shared memory. For each edge (v, u), two registers $r_{v,u}$ and $r_{u,v}$ are introduced. Node v writes to $r_{v,u}$ and reads from $r_{u,v}$ and node u writes to $r_{u,v}$ and reads from $r_{v,u}$. Direct access to variables of other nodes is not possible. Read/write-atomicity means that a move of a node must involve at most one read or one write operation but not both. This model emphasizes that it might be hard for a node v to obtain a consistent snapshot of its neighborhood prior to making a move. Consider the example that v reads $r_{u,v}$ first. Before v makes the move reading $r_{u,w}$, $u \neq w \in N(v)$, u is allowed to change the value of $r_{u,v}$. Dolev et al. show that protocols that rely on the assumption of composite atomicity can be transformed to the read/write-atomicity model using a mutual exclusion protocol they provide [DIM93].

The message-passing model constitutes a more realistic alternative to the locally shared memory model. Each communication link is modeled as a queue of messages and each node has access to functions for posting and receiving messages to resp. from any adjacent queues. The characteristics of the communication links may vary. Assumptions on whether messages may be permuted, duplicated, or dropped are often made. The capacity of the channels may be infinite or bounded by a constant. The size of messages may be bounded or unbounded. As an example of a communication link that may reorder messages, consider the case of an overlay network using UDP/IP for communication between nodes. The UDP packets which wrap the messages exchanged between the nodes may travel by different routes in the underlying network, thus arriving in an order different from the one they have been sent in. Furthermore, UDP/IP packets are in general not guaranteed to arrive and may be dropped.

All of the above models exist in synchronous and asynchronous flavors. The communication register model with read/write atomicity somewhat resembles the message-passing model. For three representative message-passing models, we refer the reader to [Pel00, Section 2.3].

2.6 Anonymity and Node Identifiers

Node identifiers are used to model to what degree nodes are distinguishable. Whether nodes have such identifiers and to which degree the identifiers are unique is a major aspect of a distributed system. In this thesis, the identifier of a node $v \in V$ is denoted by $v.id$.