Scalable Fault-Containing Self-Stabilization in Dynamic Networks

Vom Promotionsausschuss der
Technischen Universität Hamburg-Harburg

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von

Sven Köhler

aus

Paderborn

2013

Date of Oral Examination:	October 7^{th}, 2013
Chair of Examination Board:	Prof. Dr. Ernst Brinkmeyer Institute of Optical Communication Technology Hamburg University of Technology
First Examiner:	Prof. Dr. Volker Turau Institute of Telematics Hamburg University of Technology
Second Examiner:	Prof. Dr. Christian Scheideler Department of Computer Science University of Paderborn

Abstract

Self-stabilizing distributed systems provide a high degree of non-masking fault-tolerance. They recover from transient faults of any scale or nature without human intervention. In general, however, the time needed to recover from small-scale transient faults may not differ significantly from the time needed to recover from large-scale transient faults. Bounding the impact of small-scale faults has been pursued with two independent objectives: reducing the time needed to recover from state corruptions, e.g., fault-containment, and optimizing the system's reaction upon topological changes, e.g., super-stabilization.

The objective of fault-containing self-stabilization is to limit the effects of the corruption of a single node's state to a local area, i.e., to contain the fault, and to ensure that correctness of the output is regained within constant time. Transformations that add the property of fault-containment to any silent self-stabilizing algorithm exist. However, their fault-gap, i.e., the time needed to prepare for the containment of another fault, is linear in the number of nodes. Hence, these transformations do not perform well in large networks. The root cause of this is the use of global synchronization and reset.

This thesis presents a novel scheme for local synchronization. Based on it, a new transformation for adding fault-containment to silent self-stabilizing algorithms is developed. Its fault-gap and slowdown are constant. These are major improvements over previous solutions. The effects of a state corruption are strictly limited to the 2-hop neighborhood of the fault.

Similar to previous work, the transformation creates backups of a node's local state to detect and revert state corruptions. However, the number of backups per node is reduced from $\mathcal{O}(\Delta)$ to 2 backups per node. In order to balance the number of backups stored by each node, this thesis presents a self-stabilizing algorithm for computing a placement of backups such that the standard deviation of the number of backups stored per node assumes a local minimum.

Super-stabilizing algorithms not only are self-stabilizing, but also guarantee that a safety property is satisfied while the system recovers from a topology change. This thesis introduces the notion of fault-containing super-

stabilization and presents a transformation that can be used to add fault-containment to any silent super-stabilizing algorithm. Fault-containing super-stabilizing distributed algorithms are fault-containing, super-stabilizing, and guarantee that the safety property is satisfied within constant time even if a corruption of a single node's state and a topology change occur at the same time.

The transformations and algorithms presented in this thesis all work under the most general model used in self-stabilization research: the unfair distributed scheduler. Their correctness is proven using a new technique called serialization which is introduced in this thesis. It is based on the observation that it is possible to replace the parallel execution of a distributed algorithm with a sequential execution, provided the algorithm satisfies a particular condition.

Acknowledgments

I would like to take the opportunity to express my deepest gratitude to all people who have contributed to the success of this thesis.

First and foremost, I thank my supervisor Prof. Dr. Volker Turau for the guidance, advice, and encouragement he provided. I was lucky to have a great supervisor who supported me in many ways. I also want to thank my second examiner Prof. Dr. Christian Scheideler for his efforts.

For extensive proofreading I must thank Gerry Siegemund, Myriam Farias, and Lawrie Griffiths. In addition, I would like to thank Yoshio Okamoto for pointing out the relationship of local k-placements to the convex cost integer dual network flow problem.

Contents

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

$$\textit{guard} \longrightarrow \textit{statement}; \textit{statement}; \ldots$$

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 $c \vdash expression$ 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,

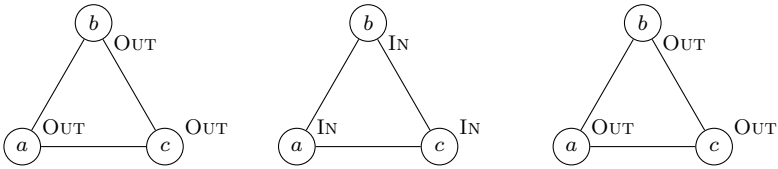


Figure 2.2: Execution of the MIS algorithm in a synchronous system

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

$$\mathit{rounds} \leq \mathit{steps} \leq \mathit{moves}$$

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 $v.id$.

In *anonymous systems*, nodes have no identifiers and thus are inherently indistinguishable. This is the weakest model. The converse model is the assumption of *globally unique* node identifiers. Each node has an identifier different from the identifier of any other node. This is the strongest possible model. Note that globally unique identifiers are not unrealistic. Any Ethernet device is given a MAC address that is unique in the world.

The requirements of several algorithms in this thesis are between these two extremes. It is required that each node can distinguish all of its neighbors. Hence, it is assumed that it holds for each node $v \in V$ that the identifiers of all neighbors $u \in N(v)$ are distinct. This model is called *weakly unique* identifiers. Furthermore, identifiers may be used for symmetry breaking between neighbors. If the identifiers are not only weakly unique, but in addition no two neighboring nodes have the same identifier, then the identifiers are said to be *locally unique*. In both cases, the identifiers are not necessarily globally unique. Locally unique identifiers are equivalent to a distance-2 vertex coloring of the graph, where the set of colors corresponds to the domain of the identifiers.

A model called port-numbering is proposed by Angluin [Ang80]. It is assumed that nodes have no identifiers and communicate via messages. However, the adjacent communication links to each node, i.e., the ports, are numbered. A node is able to retrieve the number of the link through which a message was received and can address any of the adjacent links through their number when sending messages. More formally, a *port-numbering* is a set of mappings $\{f_v \mid v \in V\}$ where each f_v is a bijective mapping from $N(v)$ to $\{1, 2, \deg(v)\}$. From the perspective of node v, $f_v(u)$ is the integer number assigned to the link (v, u). The mappings are constant but otherwise arbitrary, i.e., an algorithm must cope with any set of such mappings. In particular, the mappings f_v and f_u may not assign the same number to the link (v, u), i.e., $f_v(u) \neq f_u(v)$, and two mappings f_u and f_w, may assign a different numbers to a common neighbor v of u and w, i.e., $f_u(v) \neq f_w(v)$. The model resembles a wired network, where communication with each neighbor is done via a different wire and where each node can individually address each wire.

The idea of a port-numbering can be easily transferred to the communication register model. Each node $v \in V$ accesses any registers for communication with a neighbor $u \in N(v)$ via the port-number $f_v(u)$. However, it does not seem to have an equivalent in the shared memory model. The model is stronger than the anonymous model. For example, it is possible to count the number of neighbors with an identical local state. Assuming the shared memory model with weakly unique identifiers, it is possible to

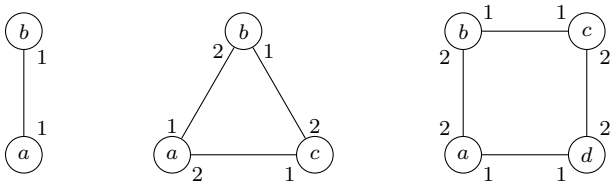


Figure 2.3: Symmetrical topologies with port-numberings

emulate message-passing with port-numbering by storing the last outgoing message for each neighbor in an array and using the identifiers as the target address of a message. However, the port-numbering model seems to be weaker than weakly unique identifiers as it is impossible for a deterministic protocol to compute weakly unique identifiers under the synchronous scheduler. Consider the topologies shown in Figure 2.3. Assuming that the communication channels do not contain any messages and that the local states of all nodes are identical, it can be shown that the local states of all nodes will remain identical under the synchronous scheduler.

2.7 Single vs. Multiple Protocols

The model described in Sections 2.2 to 2.4 differs from the model commonly used in literature on self-stabilization. The model used in this thesis allows multiple protocols per node, while the model most commonly used only permits one protocol per node. The latter can be viewed as a special case of the former. This section describes a transformation that converts any distributed algorithm $\mathcal{A}$ into an algorithm $\mathcal{A}'$ with $\mathcal{A}' = \{P'\}$ for all $v \in V$, where P' is a protocol consisting of a single rule $G_{\mathcal{A}} \to S_{\mathcal{A}}$. It is shown that for every execution of $\mathcal{A}'$ there exists an identical execution of the untransformed algorithm $\mathcal{A}$. Hence, round- and move-complexity do not increase when the transformation is applied. We conclude that all results in the multi-protocol model are transferable to the single-protocol model.

Consider that every protocol P can be written as a single rule $G_P \longrightarrow S_P$, where G_P denotes the guard and S_P denotes a sequence of statements. If P consists of multiple rules, then G_P is the disjunction of the guards of all rules of P and S_P finds the rule of P that is enabled and executes its statements. If it is permitted that multiple rules are enabled, then S_P may deal with that case as intended by the author of P.

Let $\mathcal{A}$ denote a distributed algorithm as defined in Section 2.2. Let $P_1, P_2, \ldots, P_k$ denote a sequence which contains each protocol $P_i \in \mathcal{A}(v)$ exactly once. The rule $G_{\mathcal{A}} \to S_{\mathcal{A}}$ of protocol P' is defined as follows:

$$\begin{aligned} \bigvee_{i=1}^{k} G_{P_i} \longrightarrow\ & \textbf{if } G_{P_1} \textbf{ then } S_{P_1} \\ & \textbf{if } G_{P_2} \textbf{ then } S_{P_2} \\ & \ldots \\ & \textbf{if } G_{P_k} \textbf{ then } S_{P_k} \end{aligned}$$

Instance (v, P') is enabled if and only if any $P_i \in \mathcal{A}(v)$ is enabled and a move of (v, P') executes a move of each enabled $P_i \in \mathcal{A}(v)$. In case of the distributed or synchronous scheduler, it is necessary to implement the composite atomicity described in Section 2.3. This can be done by "hiding" the changes to the local state of v made by S_{P_i} from any subsequently executed S_{P_j}, $j > i$ and the evaluation of G_{P_j}, $j > i$. The changes can be written back to the local state of v after the last move of any enabled $P_i \in \mathcal{A}(v)$. Thus, in the worst case, temporary space sufficient to store a backup of the complete local state of v is needed. In the case where multiple instances of P' on neighboring nodes are selected during the same step, the composite atomicity provided for moves of different instances takes care of "hiding" the changes on individual nodes from their neighbors until the next step.

Choosing the sequence $P_1, P_2, \ldots, P_k$ in such a way that for every P_j it holds that G_{P_j} and S_{P_j} do not read variables of any P_i, $i < j$, reduces the temporary space needed to zero. However, this is not always possible.

The following Lemmas show that $\mathcal{A}'$ is basically equivalent to $\mathcal{A}$.

Lemma 2.3. *For every execution e' of $\mathcal{A}'$ there exists an execution e of $\mathcal{A}$ such that e' is a subsequence of e.*

Proof. Let $e' = \langle c'_0, c'_1, c'_2, \ldots \rangle$ denote an execution of $\mathcal{A}'$ and $S' = \langle S'_1, S'_2, S'_3, \ldots \rangle$ the corresponding schedule.

The execution e of $\mathcal{A}$ is obtained by starting in c'_0 and using the schedule $S = \langle S_1, S_2, S_3, \ldots \rangle$. If the distributed or synchronous scheduler is assumed, then S_i is the set of all instances (v, P) enabled in c_{i-1} with $P \in \mathcal{A}(v)$ and $(v, P') \in S'_i$. In the case of the central scheduler, the scheduler S is obtained by replacing each $S'_i = \{(v, P')\}$ with a sequence of moves of all (v, P) with $P \in \mathcal{A}(v)$ in the same order that $S_{\mathcal{A}}$ executes them. □

This implies that if $\mathcal{A}$ is silent, then also $\mathcal{A}'$ is silent.

Lemma 2.4. *One round of $\mathcal{A}'$ corresponds to at least one round of $\mathcal{A}$.*

Proof. Let $e' = \langle c'_0, c'_1, \ldots, c'_l \rangle$ denote one round of $\mathcal{A}'$ and $S' = \langle S'_1, S'_2, \ldots, S'_l \rangle$ the corresponding schedule. Furthermore, let e be the execution of $\mathcal{A}$ and S be the corresponding schedule as constructed in the proof of Lemma 2.3.

The goal is to show that every (v, P_i) with $v \in V$ and $1 \leq i \leq k$ is either selected at least once in S or disabled in at least one configuration of e. If j exists such that $v \in S'_j$, then each (v, P_i), $1 \leq i \leq k$ is either an element of S_j or disabled in c_{j-1}. If $v \in V$ has not been selected in S', then v has been disabled in e' at least once. By construction of $\mathcal{A}'$, this means that (v, P_1), (v, P_2), ..., and (v, P_k) must be disabled in some configuration of e' which by Lemma 2.3 is also part of e. □

Lemma 2.5. *The transformation by which $\mathcal{A}'$ is created preserves weak fairness.*

Proof. Assume that the instance (v, P) with $v \in V$ and $P \in \mathcal{A}(v)$ is continuously enabled. The goal is to show that S_P is eventually executed by (v, P').

From (v, P) being continuously enabled it follows that (v, P') is continuously enabled and thus selected eventually by the scheduler. When selected, (v, P') executes S_P since G_P is true. □

2.8 Virtual Topologies

This thesis presents several transformations which accept a distributed algorithm $\mathcal{A}$ as their input. The transformations occasionally modify the variables of $\mathcal{A}$ in order to repair corruptions. In some situations, the transformations need to determine whether $\mathcal{A}$ is currently enabled or whether it would be enabled after modifying the variables of $\mathcal{A}$. Furthermore, some of these tests have to be performed with respect to a topology different from the current one. For this purpose we define Boolean predicates as follows:

- $G_{\mathcal{A}}(v)$: The predicate is true if and only if at least one instance (v, P) with $P \in \mathcal{A}(v)$ is enabled with respect to $\mathcal{A}$, i.e., if and only if $G_{\mathcal{A}}$ as defined in Section 2.7 is true for node v.

- $G_{\mathcal{A}}(v : x, u : y)$: The predicate is true if any only if $G_{\mathcal{A}}(v)$ is true in the virtual extended configuration $(\{(v, u)\}, c)$, where c denotes a

configuration such that local states of nodes v and u with respect to variables of $\mathcal{A}$ are equal to $x \in \sigma_{\mathcal{A}}$ and $y \in \sigma_{\mathcal{A}}$ respectively

- $\mathit{pairEnabled}_{\mathcal{A}}(v : x, u : y) \equiv G_{\mathcal{A}}(v : x, u : y) \vee G_{\mathcal{A}}(u : y, v : x)$: The predicate allows testing whether $\mathcal{A}$ has not yet terminated in a virtual topology where v is the only neighbor of u and u is the only neighbor of v.
- $\mathit{singleEnabled}_{\mathcal{A}}(v : x)$: The predicate is true if and only if $G_{\mathcal{A}}(v)$ is true in the virtual extended configuration $(\{\}, c)$, where c denotes a configuration such that the local state of node v with respect to instances of $\mathcal{A}$ is equal to $x \in \sigma_{\mathcal{A}}$.

The predicates can be automatically generated from the definition of $\mathcal{A}$ without manual reasoning.

3 Fault-Tolerance

Informally, fault-tolerance is the ability of a system to behave in a predictable way in spite of faults. Often, it also implies that the system remains operational. What behavior is desired and to what degree a system is to remain operational may vary depending on the application at hand. Hence, several types of fault-tolerance have been described in the past. This chapter gives a short overview of types of fault-tolerance and classes of faults. Sections 3.1 and 3.2 are based on the work of Tixeuil [Tix09] and Gärtner [Gä01]. The chapter concludes with the definition of a self-stabilizing distributed algorithm. Such algorithms provide a large degree of fault-tolerance.

3.1 Classification of Faults

Faults may first be classified by their nature. A fault's nature can be *state related*, *code related*, or *topology related*. A state related fault alters the values of variables of nodes in a way that can not be the result of a regular step of the distributed algorithm at hand. Such faults will be called *state corruption*. It is worth mentioning that the node identifiers are exempt from being corrupted. A code related fault results in a change of the behavior of a node, such that it deviates from the behavior implied by the distributed algorithm at hand. The most general kind are *Byzantine* faults. The local state of a Byzantine node may change arbitrarily. Another kind of code related fault is a *crash*. A crashed node stops any execution of the distributed algorithm. Topology related faults are the failure or recovery of communication links or nodes. These result in an (unexpected) change of the underlying topology graph (V, E). In this thesis, the term *topology change* will be used to refer to the removal and addition of edges. Failure and recovery of nodes are modeled by removal or addition of all edges adjacent to a node.

The time-span of a fault yields another important categorization. A *permanent* fault affects a node or communication link indefinitely. *Transient* faults affect the system for a finite amount of time, i.e., the faults eventually

stop occurring entirely. Faults which occur regularly are called *intermittent*. Such faults may stop occurring for some time but may reoccur at any time.

The memory of a node being affected by electromagnetic interference or power fluctuation is an example of a transient state corruption. Memory that is unstable due to low voltage may be regarded as an intermittent state corruption. Examples of permanent topology changes are truncated network cables, or a node with severe hardware damage. A transient topology related fault may arise from an object moving in the line of sight between two wirelessly connected nodes or a power outage that causes nodes to "disappear" for a certain amount of time. Byzantine faults, either transient or permanent, model erratic behavior of nodes as well as the scenario in which nodes are under control of an attacker with unlimited knowledge.

A third aspect of a fault is the *scale*, also called extent or span. It describes how many nodes and links are affected by the fault.

Note that a permanent small-scale topology change, like the removal of a node or a link, may just as well be regarded as a transient large-scale state corruption. This becomes apparent, as the extended configuration (E, c) can be a result of a topology change in (E', c) with $E' \neq E$ as well as the result of a state corruption in (E, c') with $c' \neq c$. Hence, from the perspective of the distributed algorithm, the nature of a fault that leads to the current configuration may not always be apparent.

3.2 Types of Fault-Tolerance

This section considers the three major types of fault-tolerance discussed in the literature. *Masking* fault-tolerance is the strongest type. Such systems continue to operate after faults without any observable loss of functionality. As an example consider a robust system as described in [Tix09], where three or more identical components perform computations on the same input redundantly. The output is processed by an additional decision unit, which determines the correct output value via a majority vote.

With other systems, the user can observe a temporary loss of functionality after a fault, but they are guaranteed to fully recover eventually. Such systems provide *non-masking* fault tolerance. *Fail-safe* systems do not necessarily recover from a fault. However, while functionality may be lost indefinitely, they guarantee their output satisfies a desired safety property.

Consider that it is in general impossible to provide any type of fault-tolerance for all classes of faults. Hence, fault-tolerance is always specified with respect to some class of faults that is assumed to be known in advance.

fault-tolerance	safety	liveness
masking	yes	yes
non-masking	no	yes
fail-safe	yes	no

Table 3.1: Three types of fault-tolerance

In the example of a robust system given above, the decision unit must never be affected by a fault.

Table 3.1 characterizes the three types of fault-tolerance concerning two properties: liveness and safety. As Gärtner puts it, liveness means that "something good eventually happens". Safety on the other hand means that "something bad never happens". Consider the following two applications: two traffic lights at a crossroad [Gä99] and spanning tree construction on a connected graph for the purpose of routing [BPBRT10]. Possible liveness properties are:

- A car, waiting in front of a red traffic light, eventually sees a green light
- The spanning tree algorithm terminates and its output is a valid spanning tree

The corresponding safety properties could look like this:

- No two traffic lights show green at the same time
- The subgraph selected by the spanning tree algorithm is acyclic (i.e., a forest)

Clearly, if both traffic lights show green at the same time, this may lead to an accident. A cycle in the output of the spanning tree algorithm may, if used for routing, lead to severe congestion as messages will move in circles. Hence, avoiding such scenarios is desirable.

Fail-safe tolerance provides safety after a fault, but no liveness. In the example of the traffic light system, this means that both traffic lights may remain red forever. Human intervention is required, for example to reboot or troubleshoot the system. Non-masking fault-tolerance does not provide safety but liveness. For the example of spanning tree construction, this would mean that the output of the algorithm may be an arbitrary subgraph. However, liveness guarantees that the output becomes a valid spanning tree

eventually, without human intervention. A masking spanning tree algorithm would provide both safety and liveness even after a fault.

The different types of fault-tolerance can be combined. A distributed algorithm can provide masking fault-tolerance with respect to one fault class while providing only fail-safe with respect to faults of a different class.

3.3 Self-Stabilization

> Such systems (with what is quite aptly called "distributed control") have been designed, but all such designs I was familiar with were not "self-stabilizing" in the sense that, when once (erroneously) in an illegitimate state, they could – and usually did! – remain so forever. (Dijkstra in [Dij74])

In 1974, Dijkstra introduced the notion of self-stabilization. As the quote above shows, he was not satisfied with the degree of fault-tolerance that distributed algorithms of his time provided. Following this motivation, he introduced the notion of self-stabilization.

Self-stabilizing distributed algorithms provide non-masking fault-tolerance, i.e., they provide liveness but no safety. In case of self-stabilization, liveness means that regardless of the initial configuration, the output of a self-stabilizing algorithm $\mathcal{A}$ is eventually correct (convergence) and it remains correct thereafter (closure). A Boolean predicate $\mathcal{L}_\mathcal{A}$ identifies configurations for which the output of $\mathcal{A}$ is considered correct. If a configuration satisfies $\mathcal{L}_\mathcal{A}$, then it is called *legitimate*. The following formalizes the definition of self-stabilization:

Definition 3.1. A distributed algorithm $\mathcal{A}$ is called *self-stabilizing* with respect to predicate $\mathcal{L}_\mathcal{A}$ if any execution of $\mathcal{A}$ satisfies the following:

- *Convergence:* A legitimate configuration is reached within a finite number of steps.
- *Closure:* $\mathcal{L}_\mathcal{A}$ is stable, i.e., any configuration subsequent to a legitimate configuration is also legitimate.

Note that convergence and closure must hold for any execution. This includes any initial configuration and any choices of the scheduler model that is used. As an example of an algorithm that has the convergence property but not the closure property, consider the algorithm $\mathcal{A}_Z$ that assigns protocol *Zero* as defined in Figure 3.1 to every node $v \in V$. We define that $\mathcal{L}_{\mathcal{A}_Z}$ is true if any only if all variables $v.s$ with $v \in V$ have the same value.

Protocol *Zero*

Nodes: v is the current node
Variables: $v.s \in \{0, 1\}$

do
$v.s \neq 0 \longrightarrow v.s := 0$
od

Figure 3.1: An algorithm satisfying convergence but not closure

Clearly, $\mathcal{A}_Z$ converges to a configuration where the local state of all nodes is zero. However, if the local state of all nodes is equal to 1 in the initial configuration, a move of a single node $u \in V$ sets $u.s$ to zero. So while $\mathcal{L}_{\mathcal{A}_Z}$ is true for the initial configuration, it becomes false due to the move of u.

Note that for any silent algorithm $\mathcal{A}$ that has the convergence property with respect to $\mathcal{L}_{\mathcal{A}}$, a predicate $\mathcal{L}'_{\mathcal{A}}$ can be constructed such that $\mathcal{A}$ is self-stabilizing with respect to $\mathcal{L}'_{\mathcal{A}}$ and $\mathcal{L}'_{\mathcal{A}} \Rightarrow \mathcal{L}_{\mathcal{A}}$ for all configurations $c \in \Sigma_{\mathcal{A}}$. The predicate $\mathcal{L}'_{\mathcal{A}}$ is defined to be true if and only if $\mathcal{A}$ has terminated. $\mathcal{A}$ terminating in a configuration that does not satisfy $\mathcal{L}_{\mathcal{A}}$ would contradict the convergence property of $\mathcal{A}$.

For measuring the time- and space-complexity of a self-stabilizing algorithm $\mathcal{A}$, the following two notions are used in this thesis:

Definition 3.2. The *stabilization time* of $\mathcal{A}$ is the worst-case number of rounds that any execution of $\mathcal{A}$ needs to reach a legitimate configuration.

The *stabilization space* of $\mathcal{A}$ is the worst-case number of bits per node that instances of $\mathcal{A}$ consume in a legitimate configuration.

Alternatively, we will also refer to the *termination time*, which denotes the worst-case number of rounds until any execution of $\mathcal{A}$ terminates. This notion applies only if $\mathcal{A}$ is silent. The stabilization and termination time may also be measured in steps or moves. The choice to measure the space required in legitimate configurations only, is motivated by the fact that the space required during stabilization may heavily depend on the initial configuration.

By guaranteeing to converge to a legitimate configuration from any initial configuration, self-stabilizing algorithms provide non-masking fault-tolerance with respect to all transient faults, regardless of their scale or nature. Also, it is well known that they converge to a legitimate configu-

ration after any permanent topology change. As discussed in Section 3.1, a topology change may also be regarded as a state corruption.

Under the unfair central scheduler, the algorithm that assigns the protocol shown in Figure 2.1 to each node $v \in V$ is in fact self-stabilizing with respect to the following predicate:

$$\forall v \in V : v.s = \textsc{Out} \Leftrightarrow \mathit{inNeighbor}(v)$$

where $\mathit{inNeighbor}(v)$ is defined in Figure 2.1. A proof for this will be discussed in Section 4.1. As pointed out in Section 2.3, the algorithm may oscillate under the distributed scheduler. A variant of the algorithm that is self-stabilizing under the distributed scheduler is discussed in Section 4.6.4.

Designing self-stabilizing distributed algorithms is non-trivial. One possible formalization of how to design a self-stabilizing algorithm $\mathcal{A}$ is discussed in [APSV91]. The proposed technique is to first choose the set of variables per node and a Boolean predicate $\mathcal{L}_{\mathcal{A}}$, such that $\mathcal{L}_{\mathcal{A}}$ is true only for configurations that are solutions to the problem at hand and such that $\mathcal{L}_{\mathcal{A}}$ is locally checkable. A predicate $\mathcal{L}_{\mathcal{A}}$ is locally checkable if it can be expressed as a conjunction of local predicates, where each local predicate is defined over the variables within $N[v]$ with $v \in V$. The second step is to design a set of rules that adjust the local states of the nodes in such a way that the local predicates are stable and gradually more and more local predicates become true. This is called local correction.

3.4 Self-Stabilization via Transformation

We are aware of several techniques to make distributed algorithms self-stabilizing via transformation. While some techniques have a considerable overhead, they prove that the requirement of self-stabilization does not render certain problems unsolvable.

The first transformation is based on the idea of periodically collecting a snapshot of the system configuration on a single node [KP93]. This node checks whether the snapshot is part of a fault-free execution of the non-stabilizing algorithm. If that is not the case, a self-stabilizing protocol performs a global reset, and the execution of the non-stabilizing algorithm starts over. The need for making a snapshot is eliminated in [APSVD94] by requiring that the predicate which asserts that the current configuration is part of a fault-free execution is locally checkable.

Two transformations, called rollback compiler and resynchronizer, are proposed in [AV91]. Both transformations require that the number of

rounds in which the input to the transformation terminates is bounded by a constant that is known a priori. Local algorithms terminate within a constant number of rounds [NS93, KMW04]. For other algorithms, where the termination time in rounds is a function $f(\Delta, n)$ that depends on n and Δ, a bound on $f(\Delta, n)$ implies a bound on Δ or n. Hence, the transformed algorithms do not scale to larger values of Δ or n.

3.5 Criticism of Self-Stabilization

Several aspects of self-stabilization have been criticized in the past. Gärtner points out that self-stabilizing protocols are rarely used in practice. He conjectures that this is due to the fact that self-stabilizing protocols only provide non-masking fault-tolerance. A similar criticism is made by Gupta [Gup97]. Indeed, the definition of self-stabilization does not specify the behavior of the distributed algorithm after the fault and prior to reaching a legitimate configuration. A self-stabilizing algorithm may therefore behave arbitrarily during that time. However, for transient faults of arbitrary scale and nature, it is impossible to provide masking fault-tolerance. Consider a state corruption of all nodes. For any non-trivial predicate $\mathcal{L}_{\mathcal{A}}$, it is possible that the values of the variables after the corruption falsify $\mathcal{L}_{\mathcal{A}}$. An attempt to combine self-stabilizing with masking fault-tolerance, with respect to a limited class of faults, was made in [GCH06]. Furthermore, we would like to point out that non-masking fault-tolerance is used in practice. For example, the spanning tree protocol specified by IEEE 802.1d implemented in most Ethernet bridges today is presumably self-stabilizing [Yam09]. The fact that the spanning tree is not intact for a certain amount of time is handled by the layers on top of Ethernet, for example TCP/IP.

Also, many refinements of self-stabilization have been proposed that aim at improving the behavior of distributed algorithms during the time immediately after a fault. These include snap-stabilization, super-stabilization, safe convergence, and fault-containment. They are discussed in more detail in Chapters 5 and 7.

Varghese points out that the state of a node is assumed to be corruptible while the program code is exempt [Var93]. He suggests that program code, which does not change frequently, can be protected from corruption by redundancy. We would like to point out that in embedded systems commonly used in sensor networks, program code and local state are kept in two different locations: ROM and RAM, respectively. In addition, some microprocessors are capable of executing the program code directly from ROM,

without copying it to RAM first. This is true at least for the ARM microprocessor architecture. Hence, especially reliable hardware can be used to store the program code while fast (but possibly more unreliable) hardware can be used to store the local state.

Furthermore, self-stabilizing algorithms do not cope well with non-transient faults, e.g., Byzantine faults [Var93]. A way to limit the impact of time-bounded Byzantine faults in self-stabilization is discussed in [YMB10]. However, for non-time-bounded Byzantine faults, the convergence property of self-stabilizing algorithms and limiting the effects of such Byzantine faults seem to be contradictory goals. As an example, consider the problem of determining the maximum degree of the network topology. The goal is to have a variable on every node that has a value equal to Δ. This is feasible in a self-stabilizing way by constructing a spanning tree [Gä03] and propagating the maximum degree to the root and then back to all nodes along the edges of the tree. However, if a node v turns Byzantine, it may behave as if an edge connects v to nodes with a degree larger than the actual value of Δ. We conclude that it is in general impossible to handle both arbitrary transient faults and limit the effect of permanent Byzantine faults for a certain class of problems.

4 Proving Self-Stabilization

> I thought that in Dijkstra 1974, I had published three solutions, but later I learned that I had also published three problems, as the programs had been given without a demonstration of their correctness. (Dijkstra in [Dij86])

Recall the definition of self-stabilization from Section 3.3. A self-stabilizing distributed algorithm provides convergence and closure for any possible execution. Hence, proving that a distributed algorithm is in fact self-stabilizing can be quite a challenge, as the proofs have to consider any corruption of the initial configuration and any possible schedule chosen by an adversary. In the case of the distributed scheduler there are up to $2^n - 1$ possible choices in each step if n nodes are enabled. Each individual choice yields a different subsequent configuration. The proofs have to cover all possible transitions.

From our experience, the proof of convergence is in general harder than the proof of closure. For silent algorithms, the proof of convergence can be divided into two parts:

- a proof of partial correctness and
- a proof that the algorithm is silent.

An algorithm is *partially correct* if it terminates only in legitimate configurations. In other words, the algorithm must not terminate prematurely, e.g., due to a deadlock. In order to prove that an algorithm is silent, one must show that every possible execution is of finite length. From the latter it follows that no livelocks can occur. Convergence follows from the algorithm being silent and partially correct. Often, a configuration is defined to be legitimate if and only if the algorithm has terminated. In this case, it is trivial to show partial correctness and closure. To show that an algorithm is silent then remains as the main challenge.

For finding a proof of convergence or for proving that a given algorithm is silent, several techniques have been proposed. This chapter gives an overview of the most common techniques. Some of them also inherently establish an upper bound for the move- or step-complexity of the algorithm at

hand. This chapter then concludes with the presentation of a new technique called serialization. It cannot be used to craft a proof of self-stabilization itself, but it allows for elevation of proofs written under the assumption of the central scheduler to the more complex case of the distributed scheduler. We show that bounds on the move- and round-complexity under the central scheduler also hold under the distributed scheduler if this technique can be applied.

4.1 Potential Functions

Potential functions (also called variant, bound, or ranking functions) were introduced as a tool for proving self-stabilization in [Kes88]. Prior to that, potential functions were used to prove termination of loops inside sequential programs [Gri81]. The relation between loop termination and a distributed algorithm $\mathcal{A}$ becomes obvious when considering the following code fragment:

while $\exists v \in V : G_{\mathcal{A}}(v)$ **do**
  execute step of $\mathcal{A}$
end

Clearly, showing that this loop terminates is equivalent to proving that $\mathcal{A}$ is silent. Recall that, depending on the scheduler at hand and algorithm $\mathcal{A}$ itself, a step of $\mathcal{A}$ can be non-deterministic. The following definition of a potential function corresponds to the one given in [Kes88]. The main idea is to find a function that decreases with every step of $\mathcal{A}$ but cannot decrease indefinitely. Provided that $\mathcal{A}$ is partially correct, the function assumes its minimal value at a legitimate configuration.

Definition 4.1. Let N denote an ordered set without an infinite strictly decreasing sequence. A function $p : \Sigma_{\mathcal{A}} \to N$ is called a *potential function* of $\mathcal{A}$ if $p(c_{i-1}) > p(c_i)$ holds for all $i > 0$ in any execution $\langle c_0, c_1, \ldots \rangle$ of $\mathcal{A}$. The value of $p(c_i)$ with $i \geq 0$ is called the *potential* of c_i.

Examples for ordered sets without an infinite strictly decreasing sequence are the set of non-negative integer numbers $\mathbb{N}_0$ or the lexicographically ordered $\mathbb{N}^x$ with $x \in \mathbb{N}$. A counter example is the set of the positive rational numbers. That set contains the infinite strictly decreasing sequence $\langle a_1, a_2, a_3, \ldots \rangle$, $a_i = 1/i$. Note that execution of $\mathcal{A}$ may terminate in configurations with non-minimal potential.

Lemma 4.2. *A distributed algorithm $\mathcal{A}$ is silent under the unfair scheduler if a potential function exists.*

The proof of the Lemma is straightforward: any execution of $\mathcal{A}$ is finite as the potential function cannot decrease indefinitely but decreases with every step of $\mathcal{A}$. The inverse also holds: if a distributed algorithm $\mathcal{A}$ is silent under the unfair distributed scheduler, then a potential function $p : \Sigma_{\mathcal{A}} \to \mathbb{N}_0$ for $\mathcal{A}$ exists. We only claim this for the case where the number of possible results of a move of any instance of $\mathcal{A}$ is finite. The proof is by construction of p. We define $p(c)$ to be the worst-case length in steps or moves of any execution of $\mathcal{A}$ starting in c. It is possible to compute this value as follows: In any configuration, a step of $\mathcal{A}$ may only yield one of a finite number of possible subsequent configurations, as the number of instances is finite and the number of possible results of each move is finite. Since $\mathcal{A}$ is silent, any possible execution terminates after a finite number of steps. Hence, the number of possible executions starting in c is finite, and an algorithm can iterate over all of them in finite time to compute the worst-case length, i.e., the value of $p(c)$.

A weaker variant of Definition 4.1 is sometimes used. It merely requires that the potential function is monotonically decreasing but not necessarily strictly decreasing with every step of $\mathcal{A}$. It must then be shown that the function decreases eventually, i.e., within finite time. In particular, this weaker definition is suitable for distributed algorithms that require a weakly fair scheduler to terminate. For such algorithms, it is safe to assume that the scheduler eventually selects the move that leads to stabilization progress. However, the number of moves executed beforehand can in general not be expressed as a function of the current configuration. Potential functions can also be used to show convergence of non-silent algorithms. In this case, potential functions are required to decrease only for any step in a non-legitimate configuration.

Any strictly decreasing potential function p with codomain $\mathbb{N}_0$ for an algorithm $\mathcal{A}$ also provides a bound on the step-complexity of $\mathcal{A}$. Recall that for the central scheduler the step-complexity is identical to the move-complexity and for the synchronous scheduler a step is identical to one round. To obtain a bound on the move complexity under the distributed scheduler, potential functions satisfying an additional requirement in addition to being strictly decreasing can be used, as the following lemma shows.

Lemma 4.3. *Let $p : \Sigma_{\mathcal{A}} \to \mathbb{N}_0$ denote a potential function such that $p(c) \geq |S| + p((c : S))$ for any configuration $c \in \Sigma_{\mathcal{A}}$ and any set $S \subseteq \mathcal{I}_{\mathcal{A}}$ of instances enabled in c. Then $p(c)$ is an upper bound on the number of moves that an execution of $\mathcal{A}$ starting c requires to reach a legitimate configuration.*

Finding a potential function for a given algorithm is a manual task and can prove to be quite challenging. A potential function is very algorithm-specific since it captures the very essence of all ideas and techniques that lead to the convergence property of the algorithm at hand [The00]. We are not aware of any fully automatic way of constructing potential functions. However, techniques which aid in the process of finding potential function are being investigated, e.g., [The00]. Also, if the potential function serves as an estimate for the time-complexity, its value is desired to be minimal, which poses an additional challenge.

As an example, we define a potential function for the maximal independent set algorithm shown in Figure 2.1. Let the potential of a configuration c be defined as the following vector of the lexicographically sorted $\mathbb{N}_0^2$:

$$\begin{pmatrix} \#\, v \in V : v.s = \textsc{In} \wedge \mathit{inNeighbor}(v) \\ \#\, v \in V : v.s = \textsc{Out} \wedge \neg \mathit{inNeighbor}(v) \end{pmatrix}$$

where $\#\, x \in X : p(x)$ denotes the number of elements x of set X that satisfy the Boolean predicate $p(x)$. Recall that the algorithm is self-stabilizing under the central scheduler. Hence, only the impact of a single instance's move on the potential vector has to be considered. An execution of Rule M1 by any node $v \in V$ decreases the second component of the vector by at least 1. It does not increase the first component, since Rule M1 is only enabled if all neighbors of v have the state Out. Rule M2 on the other hand decreases the first component by at least 1. It can increase the second component of the vector, since nodes in state Out lose a neighbor in state In. However, due to the lexicographical sorting, the vector as a whole decreases.

4.2 Convergence Stairs

Convergence stairs have been introduced in [GM91]. They formalize the intuition that there are certain observable "milestones" in the convergence process of a self-stabilizing distributed algorithm $\mathcal{A}$. A convergence stair consists of a series of Boolean predicates, each of them describing an individual milestone of the convergence process.

Definition 4.4. A *convergence stair* of $\mathcal{A}$ is a sequence $\langle R_1, R_2, \ldots, R_i \rangle$ of Boolean predicates over $\Sigma_{\mathcal{A}}$ with $R_1 = \text{true}$, which satisfy the following requirements for any execution of $\mathcal{A}$:

- **Closure:** each R_j, $j = 1, 2, \ldots, i$ is stable.

- **Convergence:** if R_{j-1} is satisfied, then a configuration satisfying R_j is reached within a finite number of steps, for $j = 2, 3, \ldots, i$.

Lemma 4.5 ([GM91, Theorem 7]). *A convergence stair $\langle R_1, R_2, \ldots, R_i \rangle$ of a distributed algorithm $\mathcal{A}$ exists if and only if $\mathcal{A}$ is self-stabilizing with respect to R_i.*

Clearly, if $\mathcal{A}$ is self-stabilizing with respect to $\mathcal{L}_\mathcal{A}$, then $\langle true, \mathcal{L}_\mathcal{A} \rangle$ is convergence stair of $\mathcal{A}$. This proves the necessity of a convergence stair. To show that the existence of a convergence stair is sufficient for self-stabilization, consider the following: The initial configuration of any execution satisfies R_1 since $R_1 = \text{true}$. The convergence property of the convergence stair guarantees that any execution of $\mathcal{A}$ reaches a configuration satisfying R_2 in finite time, then a configuration satisfying R_3 in finite time, and so forth. Eventually, any execution reached a configuration satisfying R_i. Since R_i is stable, $\mathcal{A}$ is self-stabilizing with respect to R_i.

Convergence stairs are a good tool to break down the proof of self-stabilization into several parts. The motivation for doing so is that the proofs for each step of the stair, i.e., convergence from R_{j-1} to R_j and the closure of R_j, may be simpler than a proof of self-stabilization in one piece. However, each step of the stair may require an individual technique for the proof. Convergence stairs do not inherently result in a known bound for the time-complexity of the algorithm at hand, unless a bound for the convergence time of each stair has been established.

Convergence stairs and potential functions are somewhat related. Certainly, convergence stairs can be constructed from a potential function p by defining $R_i(c) \equiv p(c) \leq p_i$, where $\langle p_i \rangle_i$ is a decreasing sequence of potentials. Potential functions and convergence stairs are also sometimes combined. Consider a potential function whose value does not decrease with every step. Instead, it is shown that the potential never increases (closure) and that the potential decreases after a finite number of steps for any execution of $\mathcal{A}$ (convergence). Each potential value can be seen as a step of a convergence stair.

4.3 Graph Reduction

For a given distributed algorithm $\mathcal{A}$, it may be possible to establish a relation between the length of executions for a topology $G = (V, E)$ and a topology $G' = (V', E')$, where G' is obtained from G by removing nodes or edges. Then, a bound on the length of executions can be derived via induction over the number of nodes or the number of edges in the graph.

In [TH11], this technique is applied to the weighted matching algorithm given in [MM07]. Turau and Hauck establish a relation between the length of executions for a graph G and executions for the graph G' which is obtained by removing one edge from G. Via induction over the number of edges, they are able to show that the algorithm stabilizes in $\mathcal{O}(nm)$ moves. The technique is applied to executions under the central scheduler as well as the distributed scheduler. The obtained bounds are a considerable improvement over the bound $\mathcal{O}(2^n)$ given by Manne and Mjelde, the original authors of the algorithm.

4.4 State Analysis

Depending on the algorithm and the problem that it solves, it is sometimes possible to prove termination entirely based on local observations. Such proofs can be found in [Tur07]. For both algorithms in the mentioned paper it is shown that the maximal number of state changes (and thus moves) per node is bounded by a constant. The application of this technique requires certain "locality" properties of the problem and the algorithm at hand.

As an example, we construct such a proof for the maximal independent set algorithm shown in Figure 2.1. In any execution of the algorithm under the central scheduler, a node v joins the independent set (i.e., changes its state to In) at most once. This holds since Rule M1 makes sure that all neighbors of v are in state Out. After node v changes its state to In, all neighbors will be disabled with respect to Rule M2 forever. From this we can conclude that a node leaves the independent set at most once. Hence, the number of moves per node is limited to 2.

Another technique is to prove that an algorithm $\mathcal{A}$ never enters the same configuration twice. If the set $\Sigma_{\mathcal{A}}$ is finite then distributed algorithm terminates eventually. Without any further analysis, this approach always yields a bound for the move-complexity that grows exponentially in the number of nodes. For any useful distributed algorithm $\mathcal{A}$, most of the nodes should have at least 2 different local states. Hence, the cardinality of the set of all possible configurations $\Sigma_{\mathcal{A}}$ is $\Omega(2^n)$.

This technique has been used by Srimani and Xu to prove termination of an algorithm that computes a weakly connected minimal dominating set [SX07, Corollary 2]. For this algorithm, scenarios are known for which indeed $\Omega(2^n)$ moves are required for stabilization [Hau08]. Another application of this technique can be found in [DLV11]. This paper focuses

on the round-complexity of the algorithm, hence a tighter bound on the number of moves is not investigated.

4.5 Composition

A common technique for developing self-stabilizing distributed algorithms is by composition of simpler self-stabilization distributed algorithms. Using certain types of compositions has the advantage that it requires little or no extra work to show that the composition is self-stabilizing. The composition "inherits" the self-stabilizing property from the algorithms that it is composed of. This section presents two types of compositions, for which this is true.

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ denote two distributed algorithms that are compatible in the sense that $\mathcal{A}_1(v)$ and $\mathcal{A}_2(v)$ are disjoint for all $v \in V$ and that there is no clash of variable names between instances of $\mathcal{A}_1$ and $\mathcal{A}_2$. Then the distributed algorithm that assigns $\mathcal{A}_1(v) \cup \mathcal{A}_2(v)$ to each node $v \in V$ is called the *composition* of $\mathcal{A}_1$ and $\mathcal{A}_2$. It is denoted by $\mathcal{A}_1 \cup \mathcal{A}_2$. We categorize compositions with regard to the communication between $\mathcal{A}_1$ and $\mathcal{A}_2$. First, consider a composition where there is no communication, i.e., the two algorithms are independent [DH99].

Definition 4.6. The composition $\mathcal{A}_1 \cup \mathcal{A}_2$ is called *independent* if no instance of $\mathcal{A}_1$ reads variables of $\mathcal{A}_2$ and vice versa. It is denoted by $\mathcal{A}_1 + \mathcal{A}_2$.

Given two algorithms that solve different tasks, e.g., one computes a spanning tree and the other a maximal independent set on the underlying topology graph $G = (V, E)$, the independent composition can be used to obtain an algorithm that solves both tasks. The second type of composition describes a unidirectional communication:

Definition 4.7. The composition $\mathcal{A}_1 \cup \mathcal{A}_2$ is called *collateral* if no instance of $\mathcal{A}_1$ reads variables of $\mathcal{A}_2$ and instances of $\mathcal{A}_2$ may read variables of $\mathcal{A}_1$. It is denoted by $\mathcal{A}_1 \triangleright \mathcal{A}_2$.

From our experience, the collateral composition, introduced in [Her92] and also described in [Tel00], is the most frequently used composition. It lets $\mathcal{A}_2$ work with the output of $\mathcal{A}_1$. For example, if $\mathcal{A}_2$ requires a leader, then $\mathcal{A}_2$ can be combined with any suitable leader election algorithm $\mathcal{A}_1$.

The following theorems discuss the properties of compositions of silent self-stabilizing algorithms under the unfair central or unfair distributed scheduler. Since all algorithms presented in this thesis are designed to be

silent under the unfair schedulers, these results will be most relevant for this thesis. Compositions of non-silent self-stabilizing algorithms often require an additional fairness assumption. This is discussed later in this section.

Theorem 4.8. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be silent and self-stabilizing with respect to Boolean predicates $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. Then $\mathcal{A}_1 + \mathcal{A}_2$ is silent and self-stabilizing with respect to $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_1 \vee \mathcal{L}_2$, and $\mathcal{L}_1 \wedge \mathcal{L}_2$.*

Let T_1 and T_2 denote the stabilization times of $\mathcal{A}_1$ and $\mathcal{A}_2$ in rounds respectively. Then $\mathcal{A}_1 + \mathcal{A}_2$ converges to $\mathcal{L}_1$ in T_1 rounds, to $\mathcal{L}_2$ in T_2 rounds, to $\mathcal{L}_1 \vee \mathcal{L}_2$ in $\min(T_1, T_2)$ rounds, and to $\mathcal{L}_1 \wedge \mathcal{L}_2$ in $\max(T_1, T_2)$ rounds.

Proof. Since $\mathcal{A}_1$ is silent, any execution of $\mathcal{A}_1 + \mathcal{A}_2$ in which the scheduler selects only instances of $\mathcal{A}_1$ is finite. Hence, the scheduler has to periodically select instances of $\mathcal{A}_2$. Since instances of $\mathcal{A}_2$ do not read variables of $\mathcal{A}_1$, moves of $\mathcal{A}_1$ do not impact the termination of $\mathcal{A}_2$. Hence, $\mathcal{A}_2$ terminates eventually. Analogously, $\mathcal{A}_1$ terminates eventually.

Since moves of $\mathcal{A}_2$ are unable to modify variables of $\mathcal{A}_1$, $\mathcal{L}_1$ is stable and is satisfied after at most T_1 rounds. Analogously, $\mathcal{L}_2$ is stable and is satisfied after at most T_2 rounds. From $\mathcal{L}_1$ and $\mathcal{L}_2$ being stable it follows that $\mathcal{L}_1 \vee \mathcal{L}_2$ and $\mathcal{L}_1 \wedge \mathcal{L}_2$ are stable. The claim that $\mathcal{L}_1 \vee \mathcal{L}_2$ holds after $\min(T_1, T_2)$ rounds and $\mathcal{L}_1 \wedge \mathcal{L}_2$ holds after $\max(T_1, T_2)$ rounds follows. □

From the proof, it can be concluded that the move-complexity of $\mathcal{A}_1 + \mathcal{A}_2$ is the sum of the move-complexity of $\mathcal{A}_1$ and $\mathcal{A}_2$. A similar result can be established for the collateral composition. However, it requires that $\mathcal{A}_2$ satisfies the definition of strong silence.

Definition 4.9. $\mathcal{A}_2$ is called *strongly silent* if any execution of a composition $\mathcal{A}_1 \cup \mathcal{A}_2$ that only contains moves of $\mathcal{A}_2$ is finite.

Theorem 4.10. *Let $\mathcal{A}_1$ be silent and self-stabilizing and $\mathcal{L}_1$ denote the predicate that is true if and only if $\mathcal{A}_1$ has terminated. Let $\mathcal{A}_2$ be self-stabilizing with respect to a Boolean predicate $\mathcal{L}_2$ under the assumption that $\mathcal{L}_1$ is satisfied. If $\mathcal{A}_2$ is strongly silent, then $\mathcal{A}_1 \triangleright \mathcal{A}_2$ is silent and self-stabilizing with respect to $\mathcal{L}_1$ and $\mathcal{L}_1 \wedge \mathcal{L}_2$.*

Let T_1 denote the termination time of $\mathcal{A}_1$ in rounds and T_2 denote the stabilization time of $\mathcal{A}_2$ under the assumption that $\mathcal{L}_1$ is satisfied. Then $\mathcal{A}_1 \triangleright \mathcal{A}_2$ converges to $\mathcal{L}_1$ in T_1 rounds and $\mathcal{L}_1 \wedge \mathcal{L}_2$ in $T_1 + T_2$ rounds.

Proof. Since $\mathcal{A}_2$ is strongly silent, any execution of $\mathcal{A}_1 \triangleright \mathcal{A}_2$ in which the scheduler selects only instances of $\mathcal{A}_2$ is finite. Hence, the scheduler has

to periodically select instances of $\mathcal{A}_1$. Since instances of $\mathcal{A}_1$ do not read variables of $\mathcal{A}_2$, moves of $\mathcal{A}_2$ do not impact the termination of $\mathcal{A}_1$. Hence, $\mathcal{A}_1$ terminates eventually. After $\mathcal{A}_1$ has terminated, $\mathcal{A}_2$ terminates within a finite number of moves.

Since moves of $\mathcal{A}_2$ are unable to modify variables of $\mathcal{A}_1$, $\mathcal{L}_1$ is stable and is satisfied after at most T_1 rounds. As $\mathcal{A}_1$ has terminated after T_1 rounds, $\mathcal{A}_2$ terminates within T_2 subsequent rounds. Hence, $\mathcal{L}_2$ is satisfied after at most $T_1 + T_2$ rounds. That $\mathcal{L}_1 \wedge \mathcal{L}_2$ is stable follows from the fact that no moves of $\mathcal{A}_1$ occur if $\mathcal{L}_1$ is satisfied. □

From the proof we can conclude that the worst-case move-complexity of $\mathcal{A}_1 \triangleright \mathcal{A}_2$ is the product of the worst-case move-complexities of $\mathcal{A}_1$ and $\mathcal{A}_2$. Consider the following scenario: the scheduler prefers instances $\mathcal{A}_2$ over instances of $\mathcal{A}_1$. Then the number of moves of $\mathcal{A}_2$ between two moves of $\mathcal{A}_1$ is bounded by the worst-case move-complexity of $\mathcal{A}_2$. Note that any move of $\mathcal{A}_1$ may "reset" the stabilization progress of $\mathcal{A}_2$.

The most notable difference between collateral and independent composition is that we make the assumption of strong silence of $\mathcal{A}_2$ in the collateral case. Strong silence means that any execution that contains only moves of $\mathcal{A}_2$ must be finite, even if the variables of $\mathcal{A}_1$, which serve as input to $\mathcal{A}_2$, have arbitrary values. This is by no means clear, as the silence of $\mathcal{A}_2$ may only have been shown under the assumption that the output of $\mathcal{A}_1$ is correct. As an example, assume that $\mathcal{A}_2$ is a self-stabilizing algorithm that has been designed to run on acyclic graphs. Typically, the presentation and the proofs of such an algorithm would not consider the case where the algorithm runs on a general graph. The collateral composition of $\mathcal{A}_2$ with a spanning tree algorithm can be used to run $\mathcal{A}_2$ on a general graph. However, the output of the spanning tree algorithm may contain cycles, and thus $\mathcal{A}_2$ may livelock. The unfair scheduler is not obligated to eventually select instances of the spanning tree algorithm, as long as instances of $\mathcal{A}_2$ are enabled.

Note that collateral composition lets $\mathcal{A}_2$ execute moves even though $\mathcal{A}_1$ has not terminated yet. Due to its self-stabilizing nature, $\mathcal{A}_2$ does not need to detect the termination of $\mathcal{A}_1$. It simply starts stabilizing automatically when $\mathcal{A}_1$ has terminated and $\mathcal{L}_1$ holds. However, the moves of $\mathcal{A}_2$ before $\mathcal{A}_1$ has terminated are sometimes unnecessary, as the actual stabilization of $\mathcal{A}_2$ towards $\mathcal{L}_2$ does not start before $\mathcal{L}_1$ holds. In [Her92], an alternative collateral composition was introduced denoted by $\mathcal{A}_1 \,;\, \mathcal{A}_2$. As part of the composition, all rules of all protocols of $\mathcal{A}_2$ are modified in such a way that any instance (v, P_2) with $P_2 \in A_2(v)$ is disabled if any instance (v, P_1)

with $P_1 \in \mathcal{A}_1(v)$ is enabled. This avoids unnecessary moves of $\mathcal{A}_2$ and thus unnecessary communication overhead to some degree. The round complexity of the composition is not impaired, i.e., $\mathcal{A}_1 \,;\, \mathcal{A}_2$ converges to $\mathcal{L}_1 \wedge \mathcal{L}_2$ within $T_1 + T_2$ rounds. In [DHR+11], this composition was reinvented under the name of hierarchical collateral composition. The notion of hierarchical composition is taken from [GH91].

Compositions can be combined with a fairness assumption about the scheduler [Dol00]. For example, the composition $\mathcal{A}_1 \triangleright \mathcal{A}_2$ of two silent algorithms is silent under the weakly fair scheduler, even if $\mathcal{A}_2$ is not strongly silent. The weakly fair scheduler guarantees that instances of $\mathcal{A}_1$ are selected and thus that $\mathcal{A}_1$ terminates eventually.

The above results allow us to reformulate other compositions in the literature. Consider the parallel composition as defined in [DH99] for the synchronous model. It can be described as $(\mathcal{A}_1 + \mathcal{A}_2 + \ldots + \mathcal{A}_k) \triangleright \mathcal{A}_B$, where $\mathcal{A}_B$ is an observer algorithm that uses the output of any $\mathcal{A}_i$, $i = 1, 2, \ldots, k$. Each $\mathcal{A}_i$ is especially suitable for a certain topology or scenario. $\mathcal{A}_i$ either stabilizes to valid output or eventually sets a flag on every node indicating that it cannot do so. The observer $\mathcal{A}_B$ uses the output of the algorithm $\mathcal{A}_i$ with minimal i such that $\mathcal{A}_i$ has not set its flag and all A_j, $j < i$ have set their flag. Consider the example from [DH99], where each $\mathcal{A}_i$ is a spanning tree protocol that computes a valid spanning tree if the network diameter satisfied $D \leq 2^i$ and sets its flag otherwise.

4.6 Serialization

The model of the central scheduler provides rather strong assumptions which simplify the design and proofs of self-stabilizing algorithms. On the other hand, the distributed scheduler is the more realistic model. Even though some algorithms that are designed for the central scheduler also stabilize under the distributed scheduler, the majority of algorithms do not have this property. In these cases new algorithms have to be devised or transformations have to be applied. Examples of such transformations can be found in [BDGM02] and [GT07]. They allow algorithms written for the central scheduler to run the unfair distributed scheduler. The former paper achieves this even for algorithms that require the k-fair central scheduler (for $k \geq n - 1$) while the latter requires algorithms that are designed for the unfair central scheduler.

Note that such transformations come at a considerable overhead. For the transformation given in [GT07], the move-complexity increases by a factor

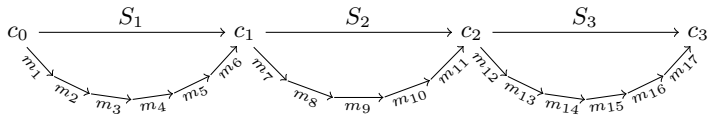


Figure 4.1: Visualization of the idea of a serialization

of $\mathcal{O}(\Delta)$. The impact on the round complexity has not been analyzed. However, we conjecture that the round-complexity increases by at least x rounds, where x is the length of the longest path with ascending node identifiers. This conjecture is based on the fact that a node waits for all nodes with a larger identifier before making a move. It is plausible that the transformation given in [BDGM02] has the same drawback since it uses identifiers for symmetry breaking. Note that x can be equal to n in the worst case. Hence, we argue that it is still desirable to write algorithms especially designed for the distributed scheduler.

A proof of self-stabilization is usually very problem-specific and does not allow for generalization. Generic proof techniques such as potential functions and convergence stairs work well for the central scheduler, since each step of the algorithm consists of a single move only. However, for the distributed scheduler the number of possible cases to consider increases dramatically. This section presents a technique for proving stabilization under the distributed scheduler by extending existing proofs that assume the central scheduler. Roughly speaking, this is done by showing that the parallel execution of moves and their sequential execution leads to the same result.

First, to simplify the reasoning, it is assumed that the distributed algorithm $\mathcal{A}$ is deterministic. Algorithms that are not deterministic are discussed in Section 4.6.6. Furthermore, the topology is assumed to be fixed.

4.6.1 Definition

To prove stabilization under the distributed scheduler of a distributed algorithm $\mathcal{A}$, we first define the notion of a serialization. A *serialization* of a set of enabled instances $S \subseteq \mathcal{I}_\mathcal{A}$ is a sequence of steps under the central scheduler such that they yield the same configuration as the single step S under the distributed scheduler. This idea is depicted in Figure 4.1: The step $c_1 = (c_0 : S_1)$ under the distributed scheduler is equivalent to several steps $(c_0 : m_1 : m_2 : \ldots : m_6)$ under the central scheduler.

Definition 4.11. Let c be a configuration and $S \subseteq \mathcal{I}_\mathcal{A}$ be a set of instances enabled in c. A sequence $s = \langle m_1, m_2, \ldots, m_k \rangle$, $m_i \in \mathcal{I}_\mathcal{A}$ is called a *serialization* of S in c if it satisfies

$$(c : S) = (c : m_1 : m_2 : \ldots : m_k) \tag{4.1}$$

S is called *serializable* in c if a serialization of S in c exists.

Assuming that instances in S alter their local state during the step $(c : S)$, then any serialization must contain each $m \in S$ at least once. Apart from that, instances may occur multiple times within the sequence. Even additional instances that are not in S may be included, but their effect has to be compensated such that Equation (4.1) holds again in the end.

Theorem 4.12. *Let c denote a configuration and $S \subseteq \mathcal{I}_\mathcal{A}$ a set of instances enabled in c. If it holds for any c and S that S is serializable in c, then for any execution e of $\mathcal{A}$ under the distributed scheduler there exists an execution e' of $\mathcal{A}$ under the central scheduler such that e is a subsequence of e'.*

The theorem can easily be proven via construction of e' by replacing each step in e with the corresponding serialization, as illustrated by Figure 4.1. However, it remains a challenge to construct such serializations or to prove that they exist for any possible step in any configuration. A technique for the construction of serialization is proposed in the next section. It is limited to serializations which include no additional instances. Such serializations have special properties as the following observation shows:

Observation 4.13. The sequence $\langle m_1, m_2, \ldots, m_k \rangle$, $m_i \in \mathcal{I}_\mathcal{A}$, is a serialization of $S \subseteq \mathcal{I}_\mathcal{A}$ in c if and only if

$$(c : S)|_{m_i} = (c : m_1 : m_2 : \ldots : m_k)|_{m_i} \qquad \forall i = 1, 2, \ldots, k$$

If $m_1, m_2, \ldots, m_k$ are distinct, then this is equivalent to

$$(c : m_i)|_{m_i} = (c : m_1 : \ldots : m_i)|_{m_i} \qquad \forall i = 1, 2, \ldots, k$$

Note that $(c : m_i)|_{m_i} = (c : S)|_{m_i}$ due to composite atomicity.

4.6.2 Construction

This section proposes a technique for constructing serializations. First, the notion of a ranking is defined. A ranking assigns a natural number

(the rank) to each enabled instance. The goal is to obtain a serialization by sorting instances by their rank. The rank may depend on the current configuration, i.e., the values of all variables, which allows the ranking to anticipate the behavior of an instance, in other words, how a move of that instance would change the variables. The section concludes with a criterion which identifies rankings that yield serializations for every S in any c for a given algorithm $\mathcal{A}$.

Definition 4.14. Let R denote an ordered set with $\bot \notin R$. The mapping $r : \mathcal{I}_\mathcal{A} \to R \cup \{\bot\}$ is called a *ranking* if the following conditions hold for any configuration:

$$r(m) = \bot \text{ if instance } m \text{ is disabled}$$
$$r(m) \in R \text{ otherwise}$$

For a given ranking it remains to show that sorting a set of instances by their rank actually yields a serialization. As a step towards this, an invariancy relation on instance/rank-tuples is defined. In general, this relation is not symmetric.

Definition 4.15. Let $r : \mathcal{I}_\mathcal{A} \to R \cup \{\bot\}$ denote a ranking. An instance m_2 of rank r_2 is called *invariant* under instance m_1 of rank r_1 if

$$(c : m_1) \vdash r(m_2) = c \vdash r(m_2)$$
$$(c : m_1 : m_2)|_{m_2} = (c : m_2)|_{m_2}$$

holds for all $c \in \Sigma_\mathcal{A}$ with $r_1 = c \vdash r(m_1)$ and $r_2 = c \vdash r(m_2)$.

If m_2 of rank r_2 is invariant under m_1 of rank r_1, then the rank of m_2 as well as the result of a move of m_2 remains the same, regardless of whether m_1 makes a move prior to m_2, or not. Note that this invariancy holds for all configurations in which m_2 and m_1 have the given ranks r_2 and r_1 respectively. The following proposition illustrates why this is useful.

Proposition 4.16. *Let c be a configuration, r a ranking, and m_1, m_2, m_3 three distinct enabled instances. If m_2 of rank $c \vdash r(m_2)$ is invariant under m_1 of rank $c \vdash r(m_1)$ and m_3 of rank $c \vdash r(m_3)$ is invariant under both m_1 of rank $c \vdash r(m_1)$ and m_2 of rank $c \vdash r(m_2)$, then $\langle m_1, m_2, m_3 \rangle$ is a serialization of $\{m_1, m_2, m_3\}$ in c.*

Proof. By Observation 4.13 it suffices to prove the following three equations:

$$(c : m_1)|_{m_1} = (c : m_1)|_{m_1} \tag{4.2}$$
$$(c : m_2)|_{m_2} = (c : m_1 : m_2)|_{m_2} \tag{4.3}$$
$$(c : m_3)|_{m_3} = (c : m_1 : m_2 : m_3)|_{m_3} \tag{4.4}$$

Equation (4.2) is clear. Equation (4.3) holds, because m_2 of rank $c \vdash r(m_2)$ is invariant under m_1 of rank $c \vdash r(m_1)$. In order to understand the validity of Equation (4.4) it is necessary to take a closer look at the sequential execution of m_1, m_2, and m_3. Consider the intermediate configuration $c' = (c : m_1)$. Because m_3 of rank $c \vdash r(m_3)$ is invariant under m_1 of rank $c \vdash r(m_1)$, it holds that $(c' : m_3)|_{m_3} = (c : m_3)|_{m_3}$. In order for Equation (4.4) to be satisfied, it must be the case that $(c' : m_2 : m_3)|_{m_3}$ equals $(c' : m_3)|_{m_3}$. This is true if m_3 of rank $c' \vdash r(m_3)$ is invariant under m_2 of rank $c' \vdash r(m_2)$. That is the case, since $c' \vdash r(m_3) = c \vdash r(m_3)$ as well as $c' \vdash r(m_2) = c \vdash r(m_2)$ and thus the invariancy of m_3 under m_2 also holds for configuration c'. □

Definition 4.17. A ranking $r : \mathcal{I}_\mathcal{A} \to R \cup \{\bot\}$ is called an *invariancy-ranking* if

$$r_2 \geq r_1 \Rightarrow \text{all instances } m_2 \in \mathcal{I}_\mathcal{A} \text{ of rank } r_2 \text{ are invariant under any instance } m_1 \in \mathcal{I}_\mathcal{A} \text{ of rank } r_1 \text{ with } m_1 \neq m_2$$

holds with respect to r for all $r_2, r_1 \in R$.

Theorem 4.18. *For an algorithm $\mathcal{A}$ with an invariancy-ranking, every set of enabled instances is serializable in any configuration.*

Proof. Let r be an invariancy-ranking, c a configuration, $S \subseteq \mathcal{I}_\mathcal{A}$ a set of instances enabled in c, and $s = \langle m_1, m_2, \ldots, m_k \rangle$ a sequence of all instances of S sorted in ascending order by their rank with respect to c. Denote by c_x the configuration $(c : m_1 : m_2 : \ldots : m_x)$ and $c_0 = c$. By Observation 4.13 it suffices to prove $c_j|_{m_j} = (c_0 : m_j)|_{m_j}$ for $j = 1, 2, \ldots, k$.

We proceed by induction on i to show that $(c_{i-1} : m_j)|_{m_j} = (c_0 : m_j)|_{m_j}$ and $c_{i-1} \vdash r(m_j) = c_0 \vdash r(m_j)$ for all $i = 1, 2, \ldots, k$ and $j = i, i+1, \ldots, k$. This is obviously true for $i = 1$. Assume the following for $i < k$:

$$c_{i-1} \vdash r(m_j) = c_0 \vdash r(m_j) \qquad \forall j = i, i+1, \ldots, k$$
$$(c_{i-1} : m_j)|_{m_j} = (c_o : m_j)|_{m_j} \qquad \forall j = i, i+1, \ldots, k$$

By assumption, m_j with rank $c_{i-1} \vdash r(m_j)$ is invariant under m_i with rank $c_{i-1} \vdash r(m_i)$ for all $j = i+1, i+2, \ldots, k$. Hence, the following is satisfied in c_i:

$$c_i \vdash r(m_j) = c_{i-1} \vdash r(m_j) = c_0 \vdash r(m_j) \qquad \forall j = i+1, i+2, \ldots, k$$
$$(c_i : m_j)|_{m_j} = (c_{i-1} : m_j)|_{m_j} = (c_0 : m_j)|_{m_j} \qquad \forall j = i+1, i+2, \ldots, k$$

In particular, it follows that $c_j|_{m_j} = (c_{j-1} : m_j)|_{m_j} = (c_0 : m_j)|_{m_j}$ for all $j = 1, 2, \ldots, k$. □

Corollary 4.19. *For any execution e under the distributed scheduler of an algorithm with an invariancy-ranking, there exists an execution e' under the central scheduler such that e is a subsequence of e'.*

Theorem 4.20. *For an algorithm $\mathcal{A}$ with an invariancy-ranking, move- and round-complexity under the central scheduler are upper bounds for the move- and round-complexity under the distributed scheduler.*

Proof. Let e be an execution of $\mathcal{A}$ under the distributed scheduler that is partitioned into rounds $r_1, r_2, \ldots, r_k$. By Corollary 4.19, an execution e' exists such that e is subsequence of e'. Let x be the number of rounds within e' and let e' be partitioned into $r'_1, r'_2, \ldots, r'_k$ such that r'_i starts with the configuration that coincides with the first configuration of r_i. Let e'_i denote $r'_i \circ r'_{i+1} \circ \ldots \circ r'_k$ and e'_{k+1} the empty sequence, where the operator $\circ$ is used to denote the concatenation of sequences. By induction over j it is shown that every e'_j is an execution of at most $x - j + 1$ rounds. It follows that e'_{k+1} is a sequence of at most $x - k$ rounds and thus $k \leq x$.

e'_1 is obviously an execution of at most x rounds. For $j \leq k$, assume that e'_j is an execution of at most $x - j + 1$ rounds. For any instance $m \in \mathcal{I}_\mathcal{A}$, r_j either contains an execution of m or a configuration in which m is disabled. Since r_j is a subsequence of r'_j. The same holds for r'_j and the first round of e'_j is a prefix of r'_j. By Lemma 2.2, e'_{j+1} consists of at most $x - j$ rounds. □

4.6.3 Partial Serialization

Serializations do not always exist. Examples of that are discussed in Section 4.6.4. However, it may be possible to "split" a step under the distributed scheduler into smaller steps that are easier to analyze. For this purpose, we introduce the concept of partial serializations, which in fact is a generalization of proper serializations.

Definition 4.21. Let c denote a configuration and $S \subseteq \mathcal{I}_\mathcal{A}$ a set of instances enabled in c. A sequence $\langle S_1, S_2, \ldots, S_l \rangle$, $S \subseteq \mathcal{I}_\mathcal{A}$ is called a *partial serialization* of S in c if it satisfies

$$(c : S) = (c : S_1 : S_2 : \ldots : S_k) \tag{4.5}$$

S is called *partially serializable* at c if a partial serialization of S in c exists.

Analogously to proper serializations, we use rankings to construct the partial serialization. The sets S_1 to S_k are obtained by sorting the instances in S by their rank. The set S_i then consists of all instances of S that have the i-th smallest rank.

Definition 4.22. Let $r : \mathcal{I}_\mathcal{A} \to R \cup \{\bot\}$ denote a ranking. An instance m_2 of rank $r_2 \in R$ is called *invariant* under a set S_1 of instances of rank $r_1 \in R$ if

$$(c : S_1) \vdash r(m_2) = c \vdash r(m_2)$$
$$(c : S_1 : m_2)|_{m_2} = (c : m_2)|_{m_2}$$

holds for all $c \in \Sigma_\mathcal{A}$ with $\forall m_1 \in S_1 : r_1 = c \vdash r(m_1)$ and $r_2 = c \vdash r(m_2)$.

Definition 4.23. A ranking $r : \mathcal{I}_\mathcal{A} \to R \cup \{\bot\}$ is called a *weak invariancy-ranking* if

$$r_2 > r_1 \Rightarrow \text{all instances } m_2 \in \mathcal{I}_\mathcal{A} \text{ of rank } r_2 \text{ are invariant under any set } S_1 \text{ of instances of rank } r_1$$

holds with respect to r for all $r_2, r_1 \in R$.

It is easy to see that every proper invariancy-ranking also is a weak invariancy-ranking.

Theorem 4.24. *For an algorithm $\mathcal{A}$ with a weak invariancy-ranking, every set of enabled instances is partially serializable in any configuration.*

Proof. Let r be a weak invariancy-ranking, c a configuration, $S \subseteq \mathcal{I}_\mathcal{A}$ a set of instances enabled in c, and $s = \langle S_1, S_2, \ldots, S_k \rangle$ a sequence of disjoint sets such that $\bigcup_{i=1}^{k} S_i = S$ and that S_i contains all instances in S with the i-th smallest rank in c. Denote by c_x the configuration $(c : S_1 : S_2 : \ldots : S_x)$ and $c_0 = c$. It suffices to prove $c_j|_m = (c_0 : m)|_m$ for all $m \in S_j$.

We proceed by by induction on $i = 1, 2, \ldots, k$ to show that $(c_{i-1} : m)|_m = (c_0 : m)|_m$ and $c_{i-1} \vdash r(m) = c_0 \vdash r(m)$ hold for all $m \in S_j$ with $j = i, i+1, \ldots, k$. This is obviously true for $i = 1$. Assume the following for $i < k$:

$$c_{i-1} \vdash r(m) = c_0 \vdash r(m) \qquad \forall m \in S_j, j = i, i+1, \ldots, k$$
$$(c_{i-1} : m)|_m = (c_o : m)|_m \qquad \forall m \in S_j, j = i, i+1, \ldots, k$$

By assumption, any $m \in S_j, j = i+1, i+2, \ldots, k$ with rank $c_{i-1} \vdash r(m)$ is invariant under S_i. Hence, the following is satisfied in c_i:

$$c_i \vdash r(m) = c_{i-1} \vdash r(m) = c_0 \vdash r(m) \quad \forall m \in S_j, j = i+1, i+2, \ldots, k$$
$$(c_i : m)|_m = (c_{i-1} : m)|_m = (c_0 : m)|_m \quad \forall m \in S_j, j = i+1, i+2, \ldots, k$$

In particular, it follows that $c_j|_m = (c_{j-1} : m)|_m = (c_0 : m)|_m$ for all $m \in S_j, j = 1, 2, \ldots, k$. □

4.6.4 Practicability and Impossibility

The overall procedure for using this technique is to first design a ranking and then to show that it is an invariancy-ranking. The design of an invariancy-ranking is a manual step and we are not aware of any way to automatize this step. Showing that a given ranking is indeed an invariancy-ranking one inspects all pairs (r_2, r_1) of ranks that satisfy $r_2 \geq r_1$. For each pair, it is then to be shown that all instances m_2 of rank r_2 are invariant under any m_1 of rank r_1. The following observation justifies that only neighboring m_2 and m_1 have to be considered in the proofs for a certain class of rankings:

Observation 4.25. Let r be a ranking, and let m_2 and m_1 be two non-neighboring instances. If $r(m_2)$ is computed solely using the values of variables of instances neighboring m_2, then instance m_2 of rank r_2 is invariant under m_1 of rank r_1 for any r_2, r_1.

Due to model constraints, the result of a move of m_2 cannot be changed by the move of m_1. So the invariancy of m_2 under m_1 only depends on the definition of r. The practicability of constructing serializations will be demonstrated in Sections 5.4.1 and 5.8.5.3. The technique is applied to two transformations to prove their correctness under the distributed scheduler.

However, invariancy-rankings do not always exist, even for algorithms that are known to stabilize under the distributed scheduler. Even worse, for some such algorithms it can be shown that serializations do not always exist. The protocol for computing a maximal independent set shown in Figure 4.2 is such an algorithm. Below, the basic obstacles that prevent the application of the new proof technique are explained. It turns out that with a few minor modifications, the algorithms can be altered so that an invariancy-ranking exists.

The protocol assigns one of the three states Out, Wait, or In to each node. If a node is in state Out and does not have a neighbor in state In, then its state is changed to Wait (Rule M1). A node in state Wait or state In changes its state to Out if it has a neighbor in state In (Rules M2

Protocol MIS_D

Nodes: v is the current node
Variables: $v.s \in \{\text{Out}, \text{Wait}, \text{In}\}$

do

[M1] $v.s = \text{Out} \wedge \neg inNeighbor(v) \longrightarrow v.s := \text{Wait}$

[M2] $\square\ v.s = \text{Wait} \wedge inNeighbor(v) \longrightarrow v.s := \text{Out}$

[M3] $\square\ v.s = \text{Wait} \wedge \neg inNeighbor(v) \wedge \neg waitNeighborWithLowerId(v) \longrightarrow v.s := \text{In}$

[M4] $\square\ v.s = \text{In} \wedge inNeighbor(v) \longrightarrow v.s := \text{Out}$

od

$$inNeighbor(v) \equiv \exists u \in N(v) : u.s = \text{In}$$
$$waitNeighborWithLowerId(v) \equiv \exists u \in N(v) : u.s = \text{Wait} \wedge u.id < v.id$$

Figure 4.2: A protocol for computing a maximal independent set [Tur07]

and M4). A node in state Wait changes its state to In if it does not have a neighbor in state In and the identifier of the node is smaller than the identifiers of all neighbors in state Wait (Rule M3).

As a first example, consider two neighboring nodes v and u, both in state In, while all their neighbors are in state Out. If the distributed scheduler selects both u and v during a single step, then both simultaneously switch their state to Out. Note that the state of u is the only reason that v is enabled and vice versa. Under the central scheduler, one of the two nodes becomes disabled after the first move and remains in state In. Hence, no serialization exists.

Now imagine that v is in state Wait and u is in state Out, while all their neighbors are in state Out. Furthermore, assume that the identifier of node u is larger than the identifier of v. If the distributed scheduler selects both v and u during a single step, then v sets its state to In by Rule M3 and u switches to state Wait by Rule M1. Again, there is no serialization because a move by v disables u and vice versa.

For the sake of optimization, [Tur07] proposes a modified version of Rule M4. The modified version only sets $v.s :=$ Out if there is an In-neighbor with a lower identifier. As a result, among any connected set of nodes in state In, the node with the lowest identifier will not change its

state. This allows serializations of the first counter-example by sorting the moves in descending order by the identifiers. The node with the lowest identifier serves as a "final cause" of the moves by the other nodes. Furthermore, it is possible to modify Rule M1 in such a way that nodes only switch from Out to Wait if there is no neighbor in state Wait with a lower identifier. Then, the second counter example becomes serializable as well. Obviously, in order for serializations to exist, situations in which moves disable moves of neighboring nodes must be avoided or at least, it must be possible to resolve these conflicts by sorting.

4.6.5 Dependency graphs

Another technique for proving that serializations exist can be found in [BGM89]. The authors define a dependency relation between instances. An instance $m_2 \in \mathcal{I}_\mathcal{A}$ of an algorithm $\mathcal{A}$ depends on $m_1 \in \mathcal{I}_\mathcal{A}$ at $c \in \Sigma_\mathcal{A}$ if there exists a set $S \subseteq \mathcal{I}_\mathcal{A}$ of instances enabled in c such that $m_1 \in S$, $m_2 \notin S$, and $(c : S : m_2)|_{m_2} \neq (c : S \setminus \{m_1\} : m_2)|_{m_2}$. In other words: m_2 depends on m_1 if the outcome of a move of m_2 varies depending on whether m_1 made a move during the previous step. The technique is applied to the three self-stabilizing algorithms from Dijkstra's original paper on self-stabilization [Dij74].

The dependency graph of configuration c is a directed graph in which instances form the nodes. The edge (m_2, m_1) exists if and only if m_2 is enabled in c and m_2 depends on m_1 in c. It is shown that if the dependency graph of a configuration c is acyclic, then a serialization for any set S of instances enabled in c exists. Like a serialization constructed via an invariancy ranking, these serializations also have the property that they contain every instance in S exactly once and no additional instances.

For all three algorithms by Dijkstra it is shown that the dependency graph for all configurations is acyclic. Since all three self-stabilizing algorithms work on a ring of n nodes, a cycle in the dependency graphs either has length n or 2 which greatly simplifies the proofs. For algorithms that run on general graphs, the absence of cycles may be harder to prove.

4.6.6 Non-deterministic protocols

The result of an instance's move may not be uniquely determined. This can be due to the use of randomization or intentional non-determinism in the protocol design. An example of the latter is a protocol for which multiple rules can be enabled at a time and for which it is specified that an enabled

rule may be chosen non-deterministically when making a move. Another example is the protocol presented in Chapter 6, where the protocol may non-deterministically choose from a set of neighbors. In such cases, a move of an instance m_2 may yield several results. The set $(c : m_2)$ describes all possible results of a move of m_2 at c. We generalize Definitions 4.11 and 4.15 for non-deterministic protocols as follows:

Definition 4.26. Let c be a configuration and $S \subseteq \mathcal{I}_\mathcal{A}$ be a set of instances enabled in c. A sequence $s = \langle m_1, m_2, \ldots, m_k \rangle$, $m_i \in \mathcal{I}_\mathcal{A}$ is called a *serialization* of S in c if it satisfies

$$(c : S) \subseteq (c : m_1 : m_2 : \ldots : m_k)$$

S is called *serializable* at c if a serialization of S in c exists.

Definition 4.27. Let $r : \mathcal{I}_\mathcal{A} \to R \cup \{\bot\}$ denote a ranking. An instance m_2 of rank r_2 is called *invariant* under instance m_1 of rank r_1 if

$$(c : m_1) \vdash r(m_2) = c \vdash r(m_2)$$
$$(c : m_1 : m_2)|_{m_2} \supseteq (c : m_2)|_{m_2}$$

holds for all $c \in \Sigma_\mathcal{A}$ with $r_1 = c \vdash r(m_1)$ and $r_2 = c \vdash r(m_2)$.

The definitions consider that a move of instance m_1 before the move of m_2 may enlarge the set of possible results of m_2's move. The proof of Theorem 4.24 can be adjusted to the definition of invariancy above and yields that an invariancy-ranking implies a serialization in the sense of Definition 4.27.

An alternative is to extend the set S with information about the result of any non-deterministic or randomized choices by the instances, such that $(c : S)$ is uniquely determined. It might also be desirable to incorporate the outcome of the non-deterministic or randomized choices in the rank of an instance.

4.7 Discussion

Using the ranking-based technique for constructing serialization presented in this chapter reduces the task of proving self-stabilization under the distributed scheduler to:

- a proof of self-stabilization under the central scheduler and
- the task of finding an invariancy-ranking.

As explained in Section 4.6.4, the proof that a given ranking is indeed an invariancy-ranking is solely based on properties of sequential executions of pairs of moves under the central scheduler. The definition of dependency as given in Section 4.6.5, on the other hand, involves steps under the distributed scheduler, which might render dependencies hard to analyze.

Algorithms exist which stabilize under the distributed scheduler, but for which it is impossible to find serializations. Hence, this technique is not always applicable. Furthermore, rankings may exist that yield serializations, but which are not invariancy-rankings. We are not aware of any examples of this. It remains to be investigated, whether it can be useful to construct serializations where instances may occur multiple times or where additional instances are inserted.

However, serialization can only be used in combination with other techniques that allow for proving self-stabilization under the central scheduler. This chapter gave an overview of the most common techniques.

5 Fault-Containing Self-Stabilization

Self-stabilizing distributed algorithms provide non-masking fault-tolerance with respect to transient faults, regardless of their scale or nature. Due to the non-masking nature of self-stabilization, there are no guarantees on the system's behavior prior to reaching a legitimate configuration. Because of that, various refinements of self-stabilization have been proposed in the past. Their aim is to either

- provide a specified behavior for the time after the fault and prior to reaching a legitimate configuration,
- minimize the time the system needs to stabilize after certain classes of faults, or
- limit the effects of the fault to a confined region close to the node affected by the fault.

Fault-containing self-stabilization, which is the focus of this chapter, is of the second and third type. Fault-containing algorithms correct their output after small-scale state corruptions in constant time.

Fault-containment is motivated by the observation that self-stabilizing algorithms do in general not take advantage of the fact that a configuration after a small-scale state corruption is "almost legitimate". In the context of fault-containment, "small-scale" specifically refers to the corruption of a single node's state. In fact, for many self-stabilizing protocols it is observed that after a state corruption, the system's configuration gradually becomes "less legitimate" before the stabilization process becomes dominant. This process is called contamination. As a consequence, large parts of the system that have not been directly affected by the fault can be impaired and may stop operating temporarily. Examples of this are given in Section 5.1. Contamination is hard to reverse. Thus, fault-containing self-stabilizing algorithms prevent contamination and contain the effect of a state corruption within the neighborhood of its occurrence.

Since small-scale faults can be assumed to occur far more often than large-scale faults, the containment of small-scale faults can greatly increase the availability of the system. In particular, fault-containing self-stabilizing

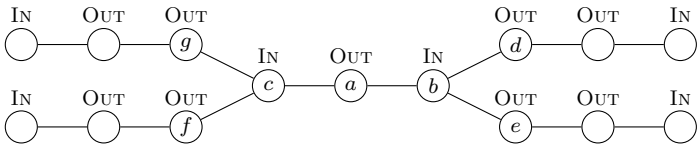


Figure 5.1: A legitimate configuration for the MIS protocol shown in Figure 2.1

algorithms retain the property of self-stabilization which guarantees convergence for any fault that cannot be contained.

The idea of dealing with faults locally was first described in [KP95, KP99] under the name of fault-local mending. However, the given algorithms are not self-stabilizing. Fault-containment transfers the idea of fault-local mending to the context of self-stabilization. The formal definition of fault-containment is given in Section 5.2. Other techniques related to state corruptions are discussed in Section 5.3. Various techniques for fault-containment are then discussed in Sections 5.4 to 5.8.

The main contribution of this chapter is the transformation given in Section 5.8, which reduces the minimum time between any two containable faults to a constant number of rounds.

5.1 Contamination

This section examines the consequences of the corruption of a single node's state if the configuration prior to the fault was legitimate. As a measure for contamination, we count the number of nodes that change their state as a result of the state corruption. It is shown that depending on the schedulers choices and the algorithm at hand, the corruption may cause state-changes of the neighborhood of the node or even the whole system.

We start with the maximimal independent set protocol *MIS* as given in Figure 2.1. Assume that there is one instance of that protocol for each node. Consider the legitimate configuration shown in Figure 5.1. The nodes in state In form a valid maximal independent set. Now consider a state corruption that sets the state of node a to In. Clearly, if the central scheduler was to select node a, it would switch back to Out via Rule M2 and the subsequent configuration would be legitimate again. In a worse-case scenario however, the scheduler selects b and c. Both nodes change their state to Out via Rule M2. Afterwards, nodes d, e, f, and g join

Protocol *BFS*

Nodes: v is the current node
Constants: $K = n$
Variables: $v.d \in \mathbb{N}$; $v.p$: Identifier of a neighbor of v

do
[B1] $v.d \neq (v.p).d + 1 \wedge (v.p).d < K \longrightarrow v.d := (v.p).d + 1$
[B2] $\square$ $(v.p).d > k.d \longrightarrow v.d := k.d + 1;\ v.p := k.id$
where k denotes a neighbor of v with $k.d = \min\{u.d \mid u \in N(v)\}$
od

Figure 5.2: A protocol for computing breadth-first spanning trees [HC92]

the independent set via Rule M1. Then the protocol terminates and the configuration is legitimate again. To summarize, in total 6 nodes changed their state as the 2-hop neighborhood of node a has been contaminated. In fact, it can be shown that for protocol *MIS* in general never more than Δ^2 nodes change their state and that only the 2-hop neighborhood of the faulty node is contaminated.

The fact that a simple algorithm for the maximal independent set problem shows such a low degree of contamination can be viewed as evidence that maximal independent set is a rather local problem. A study of the locality of problems can be found in [YT10]. The next two examples are based on algorithms for more global problems: spanning tree construction and leader election.

We choose a spanning tree algorithm that computes a breadth-first spanning tree, sometimes also called shortest path spanning tree, for the current topology. The spanning tree algorithm assumes that a designated root node $r \in V$ exists. For such a breadth-first spanning tree, the level of any node v within the tree is equal to the distance from v to r in the underlying topology $G = (V, E)$. The root node has one variable $v.d$ which always has the value 1. All other nodes $v \in V \setminus \{r\}$ execute the protocol shown in Figure 5.2. It uses two variables $v.d$ and $v.p$, where $v.d - 1$ denotes the distance of v to the root and $v.p$ is the identifier of the parent node of v within the computed tree. The term $(v.p).d$, as used by protocol *BFS*, denotes the value $u.d$ with $u \in N(v)$ and $u.id = v.p$. In a legitimate configuration, it holds for all nodes $v \in V \setminus \{r\}$ that $v.d = (v.p).d$ and that $v.p$ points to a neighbor of v with minimal distance to the root.

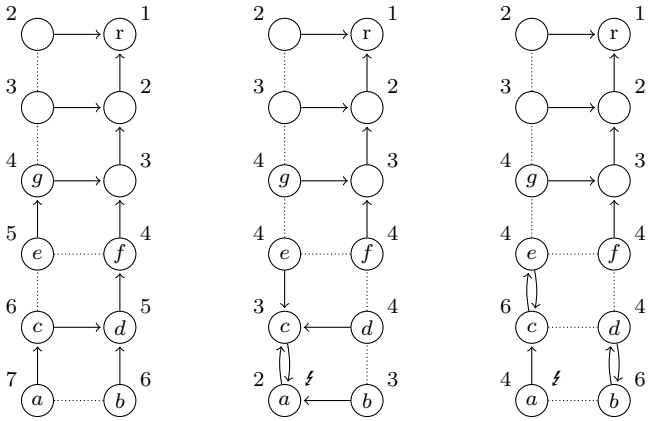


Figure 5.3: A legitimate configuration (left), maximal contamination under the central scheduler (middle), and contamination under the synchronous scheduler (right) for protocol BFS

A legitimate configuration is shown in Figure 5.3 on the left. The number next to each node denotes the value of the variable $v.d$ while the value of $v.p$ is depicted via an arrow. Dotted edges are not part of the tree. A corruption of the state of node a sets $a.d$ to the value 2. We first consider the central scheduler. Clearly, a move of (a, BFS) would yield a legitimate configuration. However, we assume that the scheduler chooses nodes b to e during the first 4 steps. Nodes b and c chose a as their parent via Rule B2. From the perspective of these nodes, it seems like there is an edge between a and r. Thus, choosing a as a parent would yield the shortest path to the root. Next, nodes d and e choose c as their parent. This yields the configuration shown in the middle of Figure 5.3. It is easy to see that for larger topologies, the contamination will affect all nodes closer to a than to r. Depending on the actual topology, the false information provided by node a may very well spread to half of the nodes of the system before the contamination process stops.

When assuming the synchronous scheduler, the result of the contamination is quite different. Again, assume that $a.d$ is corrupted to 2. During the first step, a will adjust $a.d$ to the value of 7 via Rule B1. During the same step, b and c will choose a as their parent and $b.d$ as well as $c.d$ will be set

to 3 via Rule B2. During the second step, a will set $a.d$ to 4 via Rule B1 since $c.d$ has a value of 3. Also, in our example, pairs c and e as well as b and d will choose each other as a parent via Rule B2. This will result in the configuration on the right in Figure 5.3. A wave of executions of Rule B2 will move exactly one hop further away from node a. The wave stops at nodes that are closer to r than to a.

In this example, locality, i.e., the fact that the neighbors of a cannot verify the information provided by a, is one of the reason for contamination. Note that after the state corruption, $a.p$ could also point to a non-existent neighbor of a or in particular to the root. However, in [HC92], the authors do not deal explicitly with corruptions of $a.p$. We point out that this does not impact the correctness of the algorithm, as the variable $a.p$ is never read by any neighbors of a. Illegal values of $a.p$ can be corrected with the first move of a.

A shortest path spanning tree is especially suitable for routing messages to a sink at the root node. The example under the central scheduler shows that the corruption of the state of a may result in the routing of many messages in the system towards a. Also, under the synchronous scheduler, the nodes become disconnected from the tree that is rooted at r. This will most likely result in the loss of any messages sent by a large part of the system. Hence, containment is very desirable in this scenario. It would be ideal if the corruption of $a.d$ could be repaired by a without impairing the routing of messages by its neighbors.

As a last example we choose the leader election protocol shown in Figure 5.4. It selects the node with the largest identifier as a leader. It bears great resemblence to the protocols in [LG91, AG94]. However, the one given in [LG91] is designed to work on oriented rings only and the one given in [AG94] assumes that the set of node identifiers is known to every node. The following example will consider the contamination caused by a corruption that yields a false identifier, i.e., an identifier of a non-existent node.

Each instance (v, LE) has two variables $v.m$ and $v.d$. The variable $v.m$ stores the currently selected leader. The variable $v.d$ stores the distance to the leader. A configuration is legitimate if every variable $v.m$ contains the largest identifier of the system and if $v.d$ contains the distance to the node with the identifier $v.m$ for all $v \in V$. A legitimate configuration is shown on the left in Figure 5.5. We assume that the identifiers of each node are equal to their name, e.g., the identifier of node c is the letter c. Furthermore, we assume that the identifiers are ordered alphabetically. Hence, node f has the largest identifier. Next to each node $v \in V$, the value $v.m/v.d$ is depicted.

Protocol *LE*

Nodes: v is the current node
Constants: $K = n$
Variables: $v.d \in \mathbb{N}_0$; $v.m$: Node identifier

do

[L1] $(v.d = 0 \wedge v.m \neq v.id) \vee (v.d \neq 0 \wedge v.m = v.id) \vee v.d \geq K \vee$
$v.m < v.id \vee (v.d > 0 \wedge (\nexists u \in N(v) : u.m = v.m \wedge$
$u.d + 1 = v.d)) \longrightarrow v.m := v.id;\ v.d := 0$

[L2] $\square\ \exists u \in N(v) : u.m > v.m \wedge$
$u.d + 1 < K \longrightarrow v.m := u.m;\ v.d := u.d + 1$

[L3] $\square\ (\forall u \in N(v) : u.m \leq v.m) \wedge (\exists u \in N(v) : u.m = v.m \wedge$
$u.d + 1 < v.d) \longrightarrow v.d := u.d + 1$

od

Figure 5.4: A leader election protocol for general graphs [PD02]

Consider the corruption of the state of node a such that $a.m = g$ and $a.d = 1$ holds subsequently. We discuss an execution under the central scheduler. Let the scheduler select nodes b to f in alphabetical order. All nodes make a move and execute Rule L2. The resulting configuration is depicted on the right of Figure 5.5. Eventually, the system will stabilize as the distance variables are bumped to a value larger or equal to K. If the scheduler were to first select node a, then $a.m$ and $a.d$ would be reset to a and 0 respectively via Rule L1. A subsequent execution of Rule L2 would restore $a.m$ and $a.d$ to values prior to the corruption.

As the example shows, the false identifier may be propagated to all nodes in the system. Since the nodes will attempt to communicate with a non-existent leader, the system may become dysfunctional for a certain amount of time. Again, containing the faulty information within the neighborhood of node a would be desirable, such that the remaining parts of the system can continue to operate.

5.2 Definition of Containment

This section presents metrics to measure the fault-containing capabilities of a self-stabilizing algorithm. In addition, the definition of fault-containing self-stabilization is given. Most of the definitions have been taken from

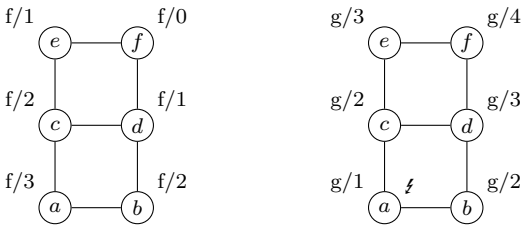


Figure 5.5: A legitimate configuration (left) and maximal contamination under the central scheduler (right) for protocol LE

[Gup97, GGHP07]. The definitions are based on categorization of variables as *primary* or *secondary*. A variable is secondary if and only if it is not primary. Informally, the set of primary variables should be chosen in such a way that all variables that constitute the output of the algorithm are primary. The secondary variables are typically helper variables. As an example, consider the variable $v.d$ used by the spanning tree protocol in Figure 5.2. This variable is required to compute a breadth-first spanning tree. However, the computed spanning tree is determined by the value of all variables $v.p$ with $v \in V$, which should thus be primary. As will become clear in this chapter, secondary variables can also help in achieving the containment of faults. The local state of a node is split into two tuples: the *primary* and the *secondary state* of the node. Similarly, a configuration c is split in two tuples c' and c'', where c' is the tuple of all primary states of all nodes and c'' is the tuple of secondary states of all nodes. c' is called *primary configuration*.

Definition 5.1. Let $\mathcal{A}$ denote a distributed algorithm that is self-stabilizing with respect to the Boolean predicate $\mathcal{L}_\mathcal{A}$. A configuration $c \in \Sigma_\mathcal{A}$ is called *k-faulty* if a configuration $c' \in \Sigma_\mathcal{A}$ satisfying $\mathcal{L}_\mathcal{A}$ exists such that c differs from c' in the local states of at most k nodes.

A k-faulty configuration with $k \ll n$ formalizes the notion of an "almost legitimate" configuration. Note that the topology, i.e., the graph $G = (V, E)$, is assumed to remain unchanged. Fault-containing self-stabilization is focused on containing the effects of the corruption of a single node's state, i.e., executions that start in a 1-faulty configuration.

Let $\mathcal{A}$ be a distributed algorithm that is self-stabilizing with respect to $\mathcal{L}_\mathcal{A}$. Furthermore, let $\mathcal{L}_\mathcal{A}^{pri}$ be a Boolean predicate over the primary variables

of $\mathcal{A}$. The truth of $\mathcal{L}_{\mathcal{A}}^{pri}$ shall imply that the output of $\mathcal{A}$ is considered correct. If $\mathcal{L}_{\mathcal{A}}^{pri}$ is satisfied, then the primary configuration is called *legitimate*. Furthermore, $\mathcal{L}_{\mathcal{A}}^{pri}$ is required to satisfy the following:

- $\mathcal{L}_{\mathcal{A}} \Rightarrow \mathcal{L}_{\mathcal{A}}^{pri}$ for all configurations $c \in \Sigma_{\mathcal{A}}$, i.e., the primary part of every legitimate configuration is also legitimate, and
- $\mathcal{L}_{\mathcal{A}}^{pri}$ is stable for any execution of $\mathcal{A}$ starting in a 1-faulty configuration.

We then define the following metrics:

Definition 5.2. The *containment time* of $\mathcal{A}$ denotes the worst-case number of rounds any execution of $\mathcal{A}$ starting at a 1-faulty configuration needs to reach a configuration satisfying $\mathcal{L}_{\mathcal{A}}^{pri}$. The *contamination number* of $\mathcal{A}$ denotes the worst-case number of nodes that change their primary state before reaching a configuration satisfying $\mathcal{L}_{\mathcal{A}}^{pri}$ in any execution of $\mathcal{A}$ starting at a 1-faulty configuration.

The contamination number is a good metric for measuring the containment of the fault in space, i.e., the impact of a fault in terms of how many nodes will be affected. The containment time on the other hand describes the containment in time, i.e., how fast the system can repair the output. However, a contamination of Δ does not allow any conclusion about the shape of the area that is affected. It can be a line of Δ nodes or nodes within the 2-hop neighborhood of the node affected by the state corruption. Hence, we occasionally use the notion of the contamination radius. If $\mathcal{A}$ has a *contamination radius* of r, then only nodes within the r-hop neighborhood of the corrupted node change their primary state in any execution of $\mathcal{A}$ starting in a 1-faulty execution. A contamination radius of 0 would imply that no contamination occurs, i.e., $\mathcal{A}$ only modifies the primary state of the corrupted node.

Definition 5.3. The self-stabilizing algorithm $\mathcal{A}$ is called *fault-containing* with respect to $\mathcal{L}_{\mathcal{A}}^{pri}$ if the containment time and the contamination number of $\mathcal{A}$ are $\mathcal{O}(1)$.

By definition, a fault-containing self-stabilizing restores correctness of its output in constant time. Also, the contamination is limited to a constant number of nodes. This implies that apart from a region of constant size, the system will continue to operate. For the leader election algorithm discussed in Section 5.1, this means that the false identifier will be quickly removed from the system.

Other definitions of fault-containment are imaginable. In Section 5.5 several algorithms are discussed that assume a weaker definition of fault-containment, where the containment time is allowed to be $\mathcal{O}(\Delta)$. However, we focus on the definitions given in [GGHP07], which match Definition 5.3.

Another important characteristic of a fault-containing algorithm is how often an algorithm can contain a fault. By definition of fault-containment, the corruption of a single node's state can only be contained if the configuration prior to the fault is legitimate. Hence, the following two metrics are defined:

Definition 5.4. The *fault-gap* of $\mathcal{A}$ denotes the worst-case number of rounds any execution of $\mathcal{A}$ starting at a 1-faulty configuration needs to reach a configuration satisfying $\mathcal{L}_{\mathcal{A}}$. The *fault-impact* of $\mathcal{A}$ denotes the worst-case number of nodes that change their local state before reaching a configuration satisfying $\mathcal{L}_{\mathcal{A}}$ in any execution of $\mathcal{A}$ starting at a 1-faulty configuration.

Informally, the fault-gap denotes the minimal distance (in time) between two containable faults. If state corruptions are expected to occur often, it is desirable that the fault-gap is small and ideally independent of the system's size. The fault-impact describes the effect of the fault on the system if both primary and secondary variables are considered. If necessary, we will refer to the *radius* of the fault-impact. If the radius of the fault-impact is r, this means that only nodes within the r-hop neighborhood of the faulty node change their local state in any execution of $\mathcal{A}$ starting in a 1-faulty configuration. In general it holds

$$\begin{aligned} \text{fault-gap} &\geq \text{containment time} \\ \text{fault-impact} &\geq \text{contamination number} \\ \text{radius of fault-impact} &\geq \text{contamination radius} \end{aligned}$$

While in principle fault-containing self-stabilization only contains the effect of the corruption of a single node's state, many of the fault-containing algorithms have been found to contain multiple state corruptions provided that the distance between any pair of affected nodes is large enough. In general, we can state that if the radius of the fault-impact is limited to r, then any number of state corruptions can be contained if the distance between any pair of affected nodes is at least $2r + 3$.

5.3 Related Concepts

This section discusses several refinements of self-stabilization. Like fault-containment, they aim at improving the behavior of self-stabilizing systems after small-scale state corruptions.

5.3.1 Safe Convergence

Safe convergence was introduced in [KM06]. While self-stabilization does not provide any form of safety during convergence, self-stabilizing algorithms that have the safe convergence property will quickly converge to a safe configuration. Afterwards, safety is maintained while the system converges to a legitimate configuration. Subsequently, whether a configuration of a distributed algorithm $\mathcal{A}$ is safe is modeled by a Boolean predicate $\mathcal{S}_\mathcal{A}$. Then the formal definition of safe convergence is as follows: a distributed algorithm $\mathcal{A}$ that is self-stabilizing with respect to $\mathcal{L}_\mathcal{A}$ has the safe convergence property with respect to the Boolean predicate $\mathcal{S}_\mathcal{A}$ if $\mathcal{S}_\mathcal{A}$ is stable for any execution of $\mathcal{A}$ and $\mathcal{L}_\mathcal{A} \Rightarrow \mathcal{S}_\mathcal{A}$ for all configurations $c \in \Sigma_\mathcal{A}$.

The definition implies that $\mathcal{A}$ is also self-stabilizing with respect to $\mathcal{S}_\mathcal{A}$. It does not specify how fast $\mathcal{A}$ should converge to $\mathcal{S}_\mathcal{A}$. However, the authors of [KM06] repeatedly stress that $\mathcal{A}$ should converge to $\mathcal{S}_\mathcal{A}$ quickly. As an example of a safely converging self-stabilizing algorithm, consider the algorithm for the minimal independent dominating set problem as given in [KM06]. The algorithm computes a minimal dominating set $D \subseteq V$ such that D is also independent. A configuration is safe if D is a valid dominating set. The algorithm reaches a safe configuration within one step under the synchronous scheduler.

Super-stabilization is a concept similar to safe convergence. A distributed algorithm is super-stabilizing if it is self-stabilizing with respect to a Boolean predicate $\mathcal{L}_\mathcal{A}$ and additionally guarantees that a safety predicate $\mathcal{S}_\mathcal{A}$ is satisfied for any execution following a single fault occurring in a legitimate configuration. Again, $\mathcal{L}_\mathcal{A} \Rightarrow \mathcal{S}_\mathcal{A}$ shall hold for all configurations $c \in \Sigma_\mathcal{A}$. Super-stabilizing algorithms thus provide safe convergence, but only for a limited class of faults. The class of faults considered depends on the application. The concept was originally introduced in [DH97] for topological faults. This variant of super-stabilization is discussed in more detail in Chapter 7. In [Her00, KUFM02] super-stabilizing algorithms are described that provide safety after a single node's state is corrupted. Since the desired safety property can easily be violated by a state corruption, the algorithms are allowed to violate it once.

Both super-stabilization and safe convergence mask the fault to some degree by providing safety. If the time needed to reach a safe configuration is much lower than the containment time of a fault-containing algorithm for the same problem, this can be a considerable advantage.

5.3.2 Adaptive Self-Stabilization

Adaptive (or time-adaptive) self-stabilization was introduced by Kutten and Patt-Shamir in [KPS99]. An adaptive self-stabilizing distributed algorithm converges from a k-faulty configuration to a legitimate primary configuration in $\mathcal{O}(k)$ rounds, i.e., the containment time for a k-faulty initial configuration is linear in k. As an example, Kutten and Patt-Shamir implement an adaptive solution to the persistent bit problem. The solution is only self-stabilizing and adaptive if $k < n/2$. The result was later improved upon: the algorithm in [BHKPS06] for the majority consensus problem, a variation of the persistent bit problem, is adaptive and fully self-stabilizing. The fault-gap of both algorithms is linear in the diameter of the network.

By sketching a proof-of-concept transformation for silent protocols, Kutten and Patt-Shamir show that time-adaptive self-stabilizing solutions exist for all non-reactive problems [KPS99]. They emphasize that the space overhead of that transformation is tremendous, and hence the transformation is to be interpreted as a possibility result. Adaptive stabilization for reactive problems is discussed in [BGK99, KPS04].

Adaptive self-stabilization can be thought of as a generalization of fault-containment. Most importantly, any adaptive self-stabilizing algorithm is also fault-containing, with the exception of the contamination number which may not be bounded by a constant.

Ghosh and He propose a methodology to design scalable self-stabilizing algorithms [GH99]. Such algorithms have a fault-gap of $\mathcal{O}(f(k))$ with respect to k-faulty initial configurations. The value of k is arbitrary and unknown a priori. If the nodes affected by the state corruption are contiguous, then $f(k) \in \mathcal{O}(k^3)$. Otherwise, $f(k)$ may be exponential in k but bounded by $\mathcal{O}(n^3)$. For $k = 1$, the containment time is clearly constant. Hence, the technique is suitable for creating fault-containing algorithms. However, for larger values of k the contamination number may grow exponentially [GH99].

Beauquier et al. [BDH06] introduce k-strong self-stabilization. A k-strong self-stabilizing protocol stabilizes from configurations that are k-faulty within $\mathcal{O}(k)$ rounds. Furthermore, strong confinement is provided, which means that all nodes that are non-faulty in the initial configuration

behave just as if the state corruption of the k faulty nodes did not happen. A transformation is given which increases the space requirements by a factor in the order of 3Δ. The input of the transformation is a silent distributed algorithm that is self-stabilizing under the distributed scheduler. The output is a 1-strong self-stabilizing algorithm that only operates under the synchronous scheduler. Transformations for larger k and providing k-strongness under the distributed scheduler are open problems.

5.3.3 Containment of Time-Bounded Byzantine Faults

As discussed in Section 3.5, Byzantine faults can contaminate the entire system with false information if the time-span of the fault is unbounded. Whether the contamination caused by Byzantine faults can be limited if the time-span is bounded is discussed in [YMB10]. A Byzantine fault is time-bounded if a Byzantine node can only make a bounded number of Byzantine state transitions. Let k denote this bound. In [YMB10] a technique called pumping is presented that limits contamination to the k-hop neighborhood of the Byzantine node. The technique is presented on a ring topology, but it is claimed that it can be extended to arbitrary topologies. The key idea of pumping is to slow down the propagation of information in such a way that a Byzantine node must perform k state transitions in order to propagate false information to nodes at distance k.

A time-bounded Byzantine fault can be seen as a sequence of state corruptions. However, these state corruptions continue as the execution of the algorithm continues. Hence, fault-containment as defined in Section 5.2 cannot deal with this type of fault.

5.4 Containment via Global Synchronization

Ghosh et al. presented the first general solution for fault-containment. They give a transformation that is able to make any silent self-stabilizing distributed algorithm fault-containing [GGHP96, GGHP07]. This section describes the transformation and then proves its correctness under the distributed scheduler. In the remainder of this section, let $\mathcal{A}$ denote a silent distributed algorithm that is self-stabilizing with respect to a Boolean predicate $\mathcal{L}_{\mathcal{A}}$. Algorithm $\mathcal{A}$ is the input to the transformation.

One essential ingredient in their transformation is a silent self-stabilizing phase clock. The implementation of the phase clock, protocol Q, is shown in Figure 5.6. The output of the transformation is the algorithm $\mathcal{A}_{FG}$ with $\mathcal{A}_{FG}(v) = \{Q\}$ for all $v \in V$. Each instance (v, Q) has a *phase counter* $v.t$

Protocol Q

Nodes: v is the current node
Constants: $M > n + 2$
Variables: $v.t$: Integer in the range $[0, M]$
Predicates:

$$raise(v) \equiv raise_1(v) \vee raise_2(v)$$
$$raise_1(v) \equiv v.t \neq M \wedge \exists u \in N(v) : v.t - u.t > 1 \wedge u.t < M - n$$
$$raise_2(v) \equiv v.t < M - n \wedge \exists u \in N(v) : u.t = M$$
$$decrement(v) \equiv decrement_1(v) \vee decrement_2(v)$$
$$decrement_1(v) \equiv v.t > 0 \wedge \forall u \in N(v) : 0 \leq v.t - u.t \leq 1$$
$$decrement_2(v) \equiv \forall u \in N(v) : v.t \geq u.t \wedge u.t \geq M - n$$
$$resetCond(v) \equiv (\forall u \in N[v] : u.t = 0) \wedge (G_{\mathcal{A}}(v) \vee \neg backupConsistent(v))$$

do

[S1] $raise(v) \vee resetCond(v) \longrightarrow v.t := M$
[S2] $\square\ decrement(v) \longrightarrow action(v.t); v.t := v.t - 1$

od

Figure 5.6: The implementation of the global phase clock as given in [GGHP07]

storing the current phase of the node. The phases range from 0 to M, where M is a constant that satisfies $M > n + 2$. During normal operation, nodes decremented their phase counters towards 0 in an even fashion until every node has reached 0. Along with this decrement, $action(v.t)$ is invoked by Rule S2. Depending on the value of $v.t$, it performs different actions. An overview of the actions is given in Figure 5.7. While being decremented, the phase counters are always kept consistent. The phase counter $v.t$ is consistent if it differs from the phase counters form the neighbors of v by at most 1.

Beside the phase counter, each (v, Q) stores the variables of $\mathcal{A}(v)$ that are defined to be primary and an array of backups. The array has one entry for each neighbor $u \in N(v)$ storing a copy of the primary variables of u. The backups and the phase counter variable are secondary. During the first 3 phases, $action(v.t)$ repairs corruptions of primary variables using backups stored on each neighbors. Moves of $\mathcal{A}$, the input to the transformation, are executed for mid-range phases. In phase 2, all backups are updated with the current values of the primary variables.

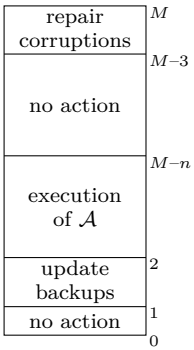


Figure 5.7: Action performed by $action(v.t)$ [GGHP07, FIGURE 5A]

Ghosh et al. assume that M is large enough for $\mathcal{A}$ to terminate within $M - n - 2$ phases. If all nodes have reached phase 0, the phase clock terminates unless the predicate $resetCond(v)$ is true for a node $v \in V$. The predicate is true if either node v is enabled with respect to $\mathcal{A}$ or if $backupConsistent(v)$ is false. The Boolean predicate $backupConsistent(v)$ is true if and only if all backups stored by (v, Q) are equal to the primary variables of the neighbors. Rule S1 ensures that a node performs a reset to phase M if an inconsistency is detected by $resetCond(v)$. The predicate *raise* ensures that the reset to M spreads to the neighbors of a node that has just performed a reset. Hence, via Rule S1, a reset spreads through the network. In a legitimate configuration, the phase counter $v.t$ is zero and $resetCond(v)$ is false for all nodes $v \in V$.

Recall that in a 1-faulty configuration, any set of variables of a single node may be corrupt. This includes, in particular, the variables introduced by the transformation such as the phase counter. The phase clock implemented by protocol Q has two special properties with regards to 1-faulty initial configurations that allow Ghosh et al. to use it for fault-containment.

- In any execution starting in a 1-faulty configuration, the phase clock initiates a global reset to phase M and
- following a reset to M, when a node executes $action(x)$, then all neighbors have executed $action(x + 1)$, for $0 \leq x < M$.

Note that there is an exception to the first property: if the corruption merely sets the phase counter of a node to 1 and no other variables are

corrupted, then the phase counter is decremented via Rule S2. This yields a legitimate configuration. Also, the second property is an implication of the consistency of the phase counters. For certain k-faulty initial configurations with $2 \leq k \leq n$, only pseudo-consistency is provided. The phase counter $v.t$ is pseudo-consistent if it is consistent or if $v.t$ and the phase counters of all neighbors of v are larger or equal to $M - n$.

The global reset to phase M ensures that corruptions are repaired before any moves of $\mathcal{A}$ are executed. Since the phase clock serves as a synchronizer, the repair of corruptions can be implemented in 3 synchronous steps. The corruption of the primary variables of a node v with $\deg(v) > 1$ is repaired during the first synchronous step. If the backups of the primary variables of v are all equal to one another but different to the primary variables of v, then the primary variable of v is overwritten with the backups. In the case where node v has only a single neighbor, i.e., $\deg(v) > 1$, only a single backup of the primary variables of v exists. Node v cannot decide without the help of u, whether the primary variables or the backups were corrupted. Node u is assumed to have more than one neighbor. Ghosh et al. do not consider the case where the graph consists of a single edge only. Informally, v informs u about being its only neighbor in step 1. In step 2, the neighbor decides whether v or u was affected by the corruption. In phase 3, v is aware of the decision of u. In addition to the repair technique described above, Ghosh et al. also describe a repair technique that takes 12 synchronous steps and does not need any backups. Instead, it temporarily collects the local states of all nodes within the 4-hop neighborhood of each node. This information is used to decide which node was affected by the state corruption. For details, we refer the reader to [GGHP07].

Consider executions starting in a 1-faulty configuration. For the faulty node and its neighbors, the global reset followed by the execution of all phases of the repair protocol is complete after a constant number of rounds. Hence, the containment time of the transformed algorithm is constant. The repair mechanism only changes the primary variables of the faulty node. Since the repair successfully restores the primary variables to values prior to the state corruption, no moves of $\mathcal{A}$ follow after the repair. Hence, the contamination number is 1. However, the global reset of the phase clock triggered by the corruption of a single node's state yields a fault-gap of $\mathcal{O}(M + D)$: The propagation of the reset takes D rounds. Subsequently, it takes M rounds to decrement the phase counter to 0. Furthermore, the fault-impact is n due to the global reset. Compared to the original protocol, the space required per node increases by a factor of Δ. Furthermore,

$\mathcal{O}(\log M)$ bits are required for the phase counter. The following theorem summarizes the results by Ghosh et al. :

Theorem 5.5. *$\mathcal{A}_{FG}$ is silent, self-stabilizing, and fault-containing with respect to $\mathcal{L}_{\mathcal{A}}$ with a contamination number of 1, a containment time of $\mathcal{O}(1)$ rounds, a fault-impact of n (radius D), and a fault-gap and stabilization time of $\mathcal{O}(M + D)$ rounds. The space required per node in a legitimate configuration is $\mathcal{O}(\log M + \Delta s_{\mathcal{A}})$ where $s_{\mathcal{A}}$ is the space required by $\mathcal{A}$ per node in a legitimate configuration.*

5.4.1 Serialization

Unfortunately, correctness of $\mathcal{A}_{FG}$ was only shown for the central scheduler in [GGHP07, Gup97]. We apply the technique of serialization, as presented in Section 4.6, to prove correctness under the distributed scheduler. $\mathcal{A}$, the input to the transformation, is assumed to be silent and self-stabilizing under the distributed scheduler. We first discuss the correctness of the phase clock only, i.e., we ignore all variables except the phase counters. To obtain serializations, the following ranking is used:

$$r(v, Q) := \begin{cases} 0 & \text{if } \mathit{decrement}(v) \\ M - v.t & \text{if } \mathit{raise}(v) \vee \mathit{resetCond}(v) \\ \bot & \text{otherwise} \end{cases}$$

Note that $\mathit{raise}(v) \vee \mathit{resetCond}(v)$ implies $v.t < M$ and thereby $r(v, Q) > 0$. So all instances of rank 0 decrement $v.t$ and all others reset it to M.

The ranking r is designed in such a way that decrements occur first within a serialization. Their order is not significant, as it will be shown that a decrement move does not disable any neighboring instances. Next, all reset moves, sorted by their phase counter in descending order, occur within the serialization. The following example illustrates why this is necessary: Consider a node v which is surrounded by nodes with a phase counter equal to 0 while $v.t = M - n$. In this configuration, node v is enabled by $\mathit{raise}_1(v)$. It is possible that all neighbors of v are enabled by $\mathit{raise}_2(v)$. If the neighbors of v perform the reset prior to v, then $v.t$ becomes pseudo-consistent, and hence (v, Q) may become disabled.

Observation 5.6. $\mathit{decrement}_1(v)$ as well as $\mathit{decrement}_2(v)$ imply $v.t \geq u.t$ for all $u \in N(v)$. So if $\mathit{decrement}(v)$ and $\mathit{decrement}(u)$ hold for two neighboring nodes v and u, then $v.t = u.t$ is true.

Lemma 5.7. *The ranking r is a valid invariancy-ranking.*

Proof. Let m_2 and m_1 denote two instances (v_2, Q) and (v_1, Q) respectively and c denotes a configuration such that $r_2 = c \vdash r(m_2)$ and $r_1 = c \vdash r(m_1)$. Furthermore, c' denotes the configuration $(c : m_1)$. Instances m_2 and m_1 are assumed to be neighboring. This is justified by Observation 4.25.

Case a) $r_2 = r_1 = 0$: Then $c \vdash \mathit{decrement}(v_2)$ and $c \vdash \mathit{decrement}(v_1)$. By Observation 5.6 it follows that $c \vdash v_1.t = v_2.t$ and $c' \vdash v_1.t = v_2.t - 1$. If $c \vdash \mathit{decrement}_1(v_2)$, then $c' \vdash \mathit{decrement}_1(v_2)$. Otherwise $c \vdash (\mathit{decrement}_2(v_2) \wedge \neg \mathit{decrement}_1(v_2))$ and thus $c \vdash v_2.t > M - n$. Hence $c' \vdash v_1.t \geq M - n$ and $c' \vdash \mathit{decrement}_2(v_2)$.

Case b) $1 \leq r_2 \leq M \wedge r_1 = 0$: Because of $r_1 = 0$, $c \vdash \mathit{decrement}(v_1)$ and thus $c \vdash v_1.t > 0$ must hold. $c \vdash \mathit{resetCond}(v_2)$ is not satisfied since it requires $c \vdash v_1.t = 0$. If $c \vdash \mathit{raise}_1(v_2)$, then there exists a node $w \in N(v_2)$ with $c \vdash (v_2.t - w.t > 1 \wedge w.t < M - n)$. $c \vdash w.t < N - n$ implies $c \vdash \neg \mathit{decrement}_2(w)$ and $c \vdash w.t - v_2.t < -1$ implies $c \vdash \neg \mathit{decrement}_1(w)$. Hence $v_1 \neq w$, $c'|_w = c|_w$ and thus $c' \vdash \mathit{raise}_1(v_2)$. If $c \vdash \mathit{raise}_2(v_2)$, then $c \vdash v_2.t < M - n$ and there exists some node $w \in N(v_2)$ with $c \vdash w.t = M$. $c \vdash w.t - v_2.t > 1$ implies $c \vdash \neg \mathit{decrement}_1(w)$ and $v_2.t < M - n$ implies $v \vdash \neg \mathit{decrement}_2(w)$. Hence $v_1 \neq w$, $c'|_w = c|_w$ and thus $c' \vdash \mathit{raise}_2(v_2)$.

Case c) $1 \leq r_2 \leq M \wedge 1 \leq r_1 \leq r_2$: If $c \vdash v_2.t < M - n$, then $c' \vdash \mathit{raise}_2(v_2)$ since $c' \vdash v_1.t = M$. Otherwise $c \vdash v_2.t \geq M - n$ which implies $c \vdash \neg \mathit{raise}_2(v_2)$ and $c \vdash \neg \mathit{resetCond}(v_2)$. Thus $c \vdash \mathit{raise}_1(v_2)$. Hence there exists a node $w \in N(v_2)$ with $c \vdash (w.t < v_2.t \wedge w.t < M - n)$. From $r_1 \leq r_2$ it follows that $c \vdash v_1.t \geq v_2.t$. Hence $v_1 \neq w$ and $c'|_w = c|_w$. Hence $c' \vdash \mathit{raise}_1(v_2)$.

In all cases $c' \vdash r(m_2) = c \vdash r(m_2)$. Hence $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$ in all cases. □

By Lemma 5.7, the phase clock implementation is correct under the distributed scheduler. It remains to take a closer look at the actions performed during the decrement of the phase counter. By Observation 5.6, the phase counters are equal for any pair of neighboring nodes that decrement their phase counters simultaneously. Hence, for any pair of neighboring nodes we can conclude that any execution of moves of $\mathcal{A}$ happens strictly after any repair of corruptions. Furthermore, updating the backups happens strictly after $\mathcal{A}$ has terminated.

It remains to show that repair actions and updating the backups by two neighboring nodes in the same phase is serializable. The update of the backups is trivially serializable, as the set of variables written (the backups) and the set of variables read (the primary variables) is disjoint. The repair actions only need to be serializable if the initial configuration is 1-faulty.

In that case, the repair actions are indeed serializable, as the repair actions only modify the primary variables and backups on one node. No other variables are modified by the repair actions, beside the variable $v.n$ which is set to $\deg(v)$ in the first synchronous step, read but not modified in the second synchronous, and deleted in the third synchronous step [GGHP07, FIGURE 7].

5.5 Problem-Specific Solutions

As well as the general solutions to fault-containment discussed in Sections 5.4 and 5.8, fault-containing self-stabilizing algorithms for several specific problems have been proposed.

Ghosh et al. describe a fault-containing self-stabilizing spanning tree algorithm in [GGP96]. It is a modification of the algorithm given in [CYH91]. It computes a general spanning tree of the graph that is not necessarily a breath-first tree. The fault-containing algorithm is designed in the distance-2 model, i.e., an instance on node v can read variables within distance 2 of v. The protocol is then transformed to the standard distance-1 model via a special synchronization scheme: A node $v \in V$ uses two variables $v.q_u$ and $v.a_u$ for each adjacent edge $(v, u) \in E$. The former is used by v to query node u. The variable $u.a_v$ is used by node u to answer the query of v. Via this mechanism, node v can gain information about its 2-hop neighborhood from any neighbor $u \in N(v)$.

In a 1-faulty configuration, one of the two variables $v.q_u$ and $u.a_v$ may be corrupted – but not both. This allows the synchronization scheme to ignore any forged answers. This is crucial for the synchronization scheme in order to preserve the fault-containment property of the algorithm in the distance-2 model.

The same authors present another fault-containing self-stabilizing spanning tree algorithm in [GGP97a] and a fault-containing leader election algorithm in [GG96]. The spanning tree algorithm computes breadth-first spanning tree and is a modification of the algorithm which is shown in Figure 5.2. The leader election is based on the algorithm in [LG91] and works on oriented rings only. In both cases, the technique is similar to the one used in [GGP96]. The fault-containing breadth-first spanning tree algorithm is first designed in the distance-3 model. Then, a synchronization scheme, an extension of the one used in [GGP96], is implemented that preserves the fault-containment properties. In the case of the leader election

algorithm, the synchronization technique is similar, but much simpler, since synchronization only occurs with the predecessor in the ring.

The use of the distance-2 or -3 model is motivated by the fact that it is easier to identify false information. Consider the example in Figure 5.3. The corruption of the distance variables of a could easily be discovered by the neighbor of a as they could determine that a does not have a suitable parent node with a matching distance to the root. In Figure 5.5, the neighbors of a could ignore the false identifier g, as g is not a neighbor of a and a is the only node within $N[a]$ that has chosen g as its leader. However, also in this model, rules with the purpose of preventing contamination must not prevent convergence.

Huang presents two fault-containing self-stabilizing algorithms: one for the maximal independent set problem (with Lin) [LH03] and one for the shortest path problem [Hua06]. Both papers are similarly structured and use the same technique. First, the secondary variables are introduced and the algorithm is adjusted so that it updates them. In the case of MIS, each node in state Out has a pointer variable that points to the neighbor in state In if there is exactly one such neighbor. Then they classify 1-faulty initial configurations into several cases and show that the faulty node is uniquely determined in each case. Then they design rules for each case. Since 2-hop information is needed, they borrow the query/answer mechanism that was used by Ghosh et al. for the protocol previously discussed in this section. They use one variable $v.q$ and $v.a$ per node. The final maximal independent set algorithm has 16 rules. The fault-containing algorithm for the shortest path problem is designed in the very same fashion, even though different secondary variables are used.

The stabilization of the two spanning tree algorithms was only shown to be finite. For the leader election algorithm, it was shown that the fault-containing variant requires n moves more to stabilize than the algorithm it is based on. The contamination number, fault-impact, and fault-gap for all protocols is listed in Table 5.1. For the first three algorithms, the results in rounds were taken from the thesis [Gup97] and the results in moves were taken from the papers referenced above. For all algorithms, the containment time was not analyzed separately to the fault-gap. Hence, we assume that it is equal to the fault-gap. For the maximal independent set and shortest path algorithms, the fault-gap was only analyzed in moves. The shortest path algorithm by Huang, which could also be used to compute a BFS spanning tree, has a significantly lower fault-gap (in moves) than the BFS spanning tree algorithm that Ghosh et al. provide. From the fault-gap we

	contam. number	fault-impact	fault-gap	space per node
Leader Election	1	3 (radius 1)	$\mathcal{O}(1)$ rounds $\mathcal{O}(1)$ moves	$\mathcal{O}(\log n)$
Spanning Tree	2	$\mathcal{O}(\Delta)$ (radius 1)	$\mathcal{O}(\Delta)$ rounds $\mathcal{O}(\Delta)$ moves	$\mathcal{O}(\Delta + \log n)$
BFS Spanning Tree	1	$\mathcal{O}(\Delta^2)$ (radius 2)	$\mathcal{O}(\Delta)$ rounds $\mathcal{O}(\Delta^3)$ moves	$\mathcal{O}(\Delta + \log n)$
MIS	1	unknown	$\mathcal{O}(\Delta)$ moves	$\mathcal{O}(\log n)$
Shortest Path	1	unknown	$\mathcal{O}(\Delta)$ moves	$\mathcal{O}(\log n)$

Table 5.1: Overview of problem-specific fault-containing algorithms

conclude that the fault-impact is also $\mathcal{O}(\Delta)$ and thus lower than of the BFS spanning tree algorithm.

All algorithms presented in this section are based on some form of local synchronization to obtain 2-hop information. We conclude that this helped for developing algorithms with a low fault-impact and fault-gap which is a considerable advantage over the transformation in Section 5.4. The algorithms do not provide a constant containment time, which means that they provide a weaker form of fault-containment than required by Definition 5.3. While the techniques used by Ghosh et al. and Huang are quite systematic, they involve manual tasks like deciding on and defining additional secondary variables in the case of [LH03, Hua06]. Hence, the techniques are not suitable for an automatic transformation. In Section 5.8, an automatic transformation based on local synchronization is provided. It provides a constant fault-gap and containment time (in rounds) as well as a constant contamination number and a fault-impact with radius 2.

5.6 Probabilistic Containment

Several publications propose a probabilistic variant of fault-containment. We discuss a method based on error-detecting codes and a method developed in the context of weak stabilization.

5.6.1 Error-Detecting Codes

Herman and Pemmaraju [HP00] use error-detecting codes to add fault-containment to silent self-stabilizing algorithms. They propose a transfor-

mation for applying error-detection codes to the local state of each node $v \in V$. Let c denote a 1-faulty configuration and v the faulty node. With a probability depending on the error-detecting code and the probability of 1-faulty configurations, a corruption of the local state of v is detectable by v itself and all neighbors of v. In order to avoid contamination, the transformation ensures that all neighbors of v refrain from making any moves until the corruption of the local state of v is fixed. The transformation assumes that node v can compute a value x within one move such that overwriting the local state of v with x renders node v as well as all its neighbors disabled. Herman and Pemmaraju show that this is not possible for all distributed algorithms. In order to do that, node v would have to have access to all local states within its 2-hop neighborhood. The goal of Herman and Pemmaraju is to avoid any replication overhead. Hence, the neighbors of v don't store copies of the local states of their neighbors. However, several criteria are discussed that identify distributed algorithms for which it is possible to compute x within one move of v.

For algorithms that satisfy the criteria in [HP00], a detectable corruption is repaired within a single step. Thus, the contamination number and the fault-impact are 1 and the containment time and fault-gap are 1 round.

5.6.2 Weak Stabilization

Weak stabilization is a relaxation of self-stabilization [Gou01]. While the requirement of closure is retained, convergence is relaxed to reachability. Recall that convergence requires that any execution reaches a legitimate configuration in finite time. Reachability only requires that for any configuration c_0 at least one execution starting in c_0 exists that reaches a legitimate configuration in a finite number of steps. In other words, there is the possibility of reaching a legitimate configuration at all times. Devismes et al. point out that weakly stabilizing solutions can be regarded as probabilistically self-stabilizing under a randomized scheduler [DTY08]. However, this only holds if the set of configurations is finite.

Dasgupta et al. have designed weakly stabilizing algorithms for the persistent-bit and the leader-election problem under the randomized central scheduler [DGX11]. The presented algorithms contain the corruption of a single node's state with a probability depending on a constant $m \in \mathbb{N}$. It is intended to be used as a tuning parameter. If the value of m is small, then the probability of containment is very low and the stabilization time is low. The larger m is, the higher the probability of containment but the

stabilization time also grows. Hence, m should be chosen such that both probability of containment and stabilization time are acceptable.

An essential part of the technique used by Dasgupta et al. is as follows: A counter variable $v.x \in \mathbb{N}$ is added to the local state of each node $v \in V$. To slowdown the contamination process, each rule $G_R \longrightarrow S_R$ that can cause contamination is split into two new rules R and R'. Rule R is enabled if G_R is satisfied and a neighbor with a larger counter value exists. When executed, rule R increments the counter $v.x$ by 1. Rule R' is enabled if G_R is satisfied and if no neighbor with a larger counter value exists. Rule R executes S_R and sets $v.x$ to the value $m+\max\{u.x \mid u \in N(v)\}$. Beside this, Dasgupta et al. add further rules in order to quickly repair corruptions. As a result, if $m \geq 2$, then the statement S_R of rules that cause contamination is executed less often than the statements of other rules. This prevents contamination with a probability depending on m.

For the persistent bit protocol, it is shown that the fault-gap, and thus the containment time, is $\mathcal{O}(\Delta^3)$ if $m >> \Delta$ and that the contamination number approaches 1 for $m \to \infty$. For the fault-gap and the containment time, it is merely shown that they are independent of n. The contamination number and the radius of the fault-impact are shown to be 4 with high probability for $m \to \infty$. The expected stabilization time of both algorithms is not analyzed.

We simulated both algorithms in a self-written simulator that implements the locally shared memory model. For the simulation, a randomized central scheduler that chooses any enabled process with equal probability was used. As a topology, a line graph with 10 nodes was used. The variables were initialized with random values, where the unbounded counters were initialized with values between 0 and 100. As a random number generator we used the Mersenne Twister from the Uncommons Maths Java library [MN98]. Our simulations unveiled that both the persistent-bit and the leader election protocol did not always converge. Especially for large m, executions of the algorithms diverged frequently. This observation is consistent with the results in [DGX11, Figure 1]. For diverging executions, the probability that the algorithms eventually stabilize would actually degrade progressively as the execution continued. The core problem is that the difference between counters of neighboring nodes v and u may grow indefinitely.

A diverging execution with $m = 3$ of the persistent bit protocol with unbounded counters is depicted Figure 5.8. The number below the nodes denotes the current bit value and the value above each node denotes the value of the counter. Throughout the execution, nodes e and f would switch their bit from 0 to 1. After about 100 rounds, their counters would outgrow

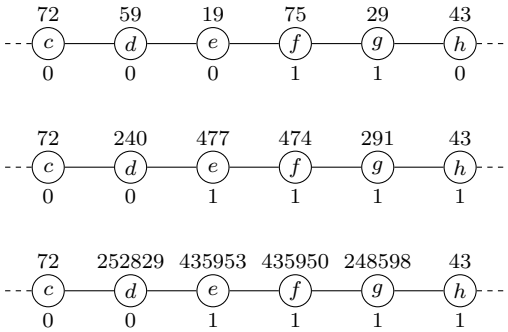


Figure 5.8: Example of a diverging execution: initial configuration (top), configuration after 10^3 rounds (middle), and configuration after 10^6 rounds (bottom)

the counters of d and g. We observed that the counter of e and f would consistently be about twice as large as the counters of d and g for large parts of the execution. After round 10^6, nodes d and g would have to increment their counters more than 200 thousand times in order to execute a move that would change the value of their bit. We conjecture that the problems arise from the use of unbounded counters. As the set of configurations is infinite, the results in [DTY08] do not apply.

A variant of the weakly stabilizing solution to the persistent-bit problem using bounded counters was proposed by Dasgupta et al. in [DGX07]. For this algorithm, the tuning parameter M also defined the range of the counter. Consistent with the results in [DTY08], the algorithm stabilized under the randomized central scheduler in all simulations.

We conclude that practical relevance of algorithms that use an unbounded counter is questionable. While it has been shown of both that they are weakly stabilizing, we argue that weak stabilization does not always result in some form of stabilization in practice. One could easily change the rules in such a way that a counter $v.x$ is bumped to at least $\max\{u.x \mid u \in N(v)\} - m$ before each increment operation. However, how this impairs the fault-containment of the algorithms is an open problem.

5.7 Priority Scheduling

In a case study by Ghosh and He, the priority scheduling model is used to design a fault-containing self-stabilizing spanning tree algorithm [GH00]. In the priority scheduling model, a priority is assigned to each rule of a protocol. The model guarantees that a node $v \in V$ executes a rule with a priority p only if no rule with a priority higher than p is enabled on node v or its neighbors. Ghosh and He point out that the priority model greatly simplified the design of the spanning tree algorithm and the proof of fault-containment.

The motivation for this model is that it is often possible to instantly repair the state corruption in a 1-faulty configuration by a move of the faulty node. This is also true for the examples given in Section 5.1. However, nodes neighboring the faulty node may make moves first which leads to contamination. This can be circumvented by prioritizing rules that successfully repair the corruption over rules that lead to contamination. In general, however, this might not always be possible without impairing the self-stabilization property of the algorithm. In order to construct the fault-containing spanning tree protocol based on the priority scheduling model, Ghosh and He add several rules with different priorities to the spanning tree algorithm of [CYH91]. They show that the modified algorithm is self-stabilizing and fault-containing in the priority scheduler model.

As it is non-trivial to implement such a priority scheduler in the standard model, Ghosh and He give an implementation of the priority scheduler in [GH00]. They show that their implementation satisfies the constraints of the priority scheduling model even if the initial configuration is 1-faulty. Hence, the combination of the priority scheduler and the fault-containing spanning tree protocol remains fault-containing.

The fault-gap, and thus the containment time of the algorithm, are constant in the standard model. However, the stabilization time of the fault-containing algorithm for arbitrary initial configurations is not analyzed. Also, it is an open problem whether the priority scheduling approach could be used to design a transformation for fault-containment.

The priority scheduler seems to cause a considerable slowdown. Ghost and He show that every 4 rounds, the priority scheduler executes at least one move of the protocol that runs on top of it. Under the assumption that a protocol executes $\mathcal{O}(n)$ moves per round, this would mean a slow-down factor of $\mathcal{O}(n)$. We proceed to give an example for which this slow-down is actually reached.

Consider the standard maximal independent set protocol for the central scheduler as given in Figure 2.1. It stabilizes in 2 rounds: In the first round, nodes leave the independent set, until no conflicts exist. In the second round, all nodes that do no have a neighbor in the independent set join it. No conflicts are created. Hence, the algorithm terminates. The same priority is assigned to all rules of the MIS protocol. As a topology, we chose a line graph, where the Ids are strictly ascending in one direction. The state of all nodes is assumed to be equal to In initially.

When the MIS protocol is executed with the priority scheduler implementation provided by Ghosh and He, then each node waits before executing a move of the protocol until all neighbors of a node point towards it. A node always points towards its neighbor with the largest priority or, if all priorities are equal, to the node with the lowest Id. After a constant number of rounds, all nodes point towards the node with the lowest Id, which is then allowed to leave the independent set. The node with second smallest Id leaves the independent set after four more rounds. Note that we now disregard any moves of nodes that join the independent set, which would further delay the moves of the nodes leaving the independent set. So we assume that the node with the third smallest Id is able to leave the independent set after the eighth round. This continues until $n - 1$ nodes have left the independent set at least once. This is the case after $4n + \mathcal{O}(1)$ rounds.

5.8 Containment via Local Synchronization

This section describes a new transformation that adds fault-containment to any given silent self-stabilizing distributed algorithm $\mathcal{A}$. The result of the transformation is algorithm $\mathcal{A}_{FL}$, which is silent, self-stabilizing, and also fault-containing. The fault-gap of $\mathcal{A}_{FL}$ is constant. This is the first transformation for fault-containment to achieve this. In addition, the slowdown is constant, i.e., if $\mathcal{A}$ terminates within T rounds, then $\mathcal{A}_{FL}$ terminates in $\mathcal{O}(T)$ rounds. Furthermore, the transformation is designed to work under the most general scheduler: Algorithm $\mathcal{A}_{FL}$ is fault-containing self-stabilizing under the unfair distributed scheduler (resp. central scheduler) if $\mathcal{A}$ is silent self-stabilizing under the unfair distributed scheduler (resp. central scheduler).

Redundancy is used to detect and quickly repair state corruptions. Each node holds a backup of the primary variables of all its neighbors. This is similar to what has been done in [GGHP07]. Moreover, the transformation is based on the concept of cells. A cell spans across a node and all its

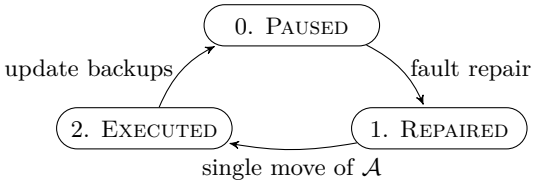


Figure 5.9: States and transitions of a cell

neighbors. Cells can be viewed as a local phase clock which provides a form of local synchronization. Cells allow us to achieve the following objectives without global synchronization:

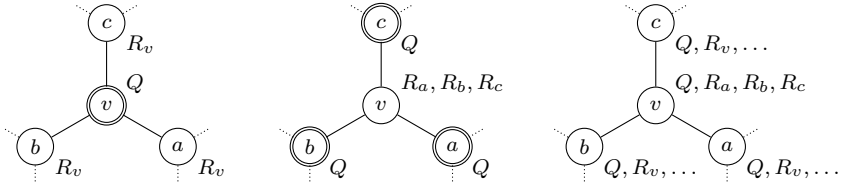
- *No contamination*: For any execution starting in a 1-faulty configuration, the neighbors of the faulty node must not execute algorithm $\mathcal{A}$ until the primary variables of the faulty node have been repaired. Otherwise, the execution of $\mathcal{A}$, which is assumed to cause contamination, would let faulty information spread to the neighbors and beyond. This contamination process is hard to reverse.

- 2-*hop knowledge for repair*: In some cases, deciding whether the primary variables or the backups have been corrupted is not possible without information contributed by neighboring nodes. For any execution starting in a 1-faulty configuration, it must be guaranteed that all neighbors update the variables through which they provide this information before a node attempts to repair its primary variables.

We proceed to first explain the concept of a cell and how the fault detection is implemented on top of it.

5.8.1 Cells

A basic idea of cells is captured in Figure 5.9. Each cell is an implementation of the depicted state-machine consisting of 3 states and 3 transitions. Roughly speaking, a cell repairs corruptions using the backups during the first transition. During the second transition, moves of algorithm $\mathcal{A}$ are executed. The backups are updated during the third transition. How this is implemented is described in detail Section 5.8.3. Hereafter, the states of the state-machine are called *positions* to avoid confusion with the notion of local states.



(a) cell C_v (b) cells C_a, C_b, and C_c (c) composition

Figure 5.10: The different roles of node v

Note that the implementation of a cell adds new secondary variables, e.g., to store the current position of a cell. These variables are not exempt from being corrupted. Unless appropriate precautions are taken, a corruption of a single node's local state may change the position of a cell arbitrarily. To address this, a *dialog* between a node and its neighbors is established. Each transition from one position to another requires the following three steps:

1. Ask neighbors for approval of the transition.
2. Wait for acknowledgments from all neighbors.
3. Perform the transition.

All neighbors execute the counterpart:

1. Wait for a query.
2. Respond with an acknowledgment.

This dialog suffices to achieve the previously defined objectives. We proceed to describe its implementation.

5.8.2 Cell Dialog

Each node $v \in V$ is center of a cell denoted by C_v. Cell C_v consists of the so-called *center instance* (v, Q) and the *responding instances* (u, R_v), $u \in N(v)$. Note that protocol R_v is parametrized. The parameter v specifies the neighbor of u with which (u, R_v) interacts. This scheme is illustrated in Figure 5.10a. Cells overlap and two cells C_v and C_a are said to be *neighboring* if nodes a and v are neighbors. Figure 5.10b shows node v as part of cells C_a, C_b, and C_c. All protocols executed by each node are

Protocol Q (dialog only)

Nodes: v is the current node
Variables: $v.s, v.q \in \mathbb{Z}_3$

do

[Q1] $\neg \mathit{dialogConsistent}(v) \wedge (v.s, v.q) \neq (\textsc{Paused}, \textsc{Paused}) \longrightarrow$
$(v.s, v.q) := (\textsc{Paused}, \textsc{Paused})$

[Q2] $\square\ \mathit{dialogPaused}(v) \wedge \mathit{startCond}_Q(v) \longrightarrow\ v.q := \textsc{Repaired}$

[Q3] $\square\ \mathit{dialogAcknowledged}(v) \longrightarrow$
$v.s := v.q$
if $v.s \neq \textsc{Paused}$ **then** $v.q := (v.s + 1) \bmod 3$

od

Protocol R_v (dialog only)

Nodes: u is the current node, $v \in N(u)$ is the center of the cell
Variables: $u.r_v \in \mathbb{Z}_3$

do

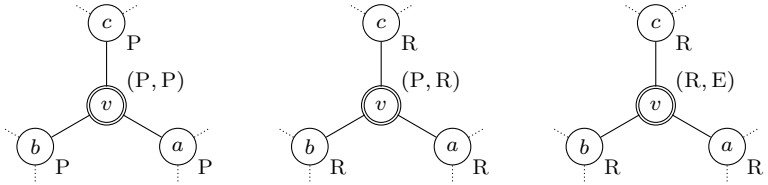
[R1] $v.s = v.q = \textsc{Paused} \wedge u.r_v \neq \textsc{Paused} \longrightarrow\ u.r_v := \textsc{Paused}$

[R2] $\square\ \mathit{validQuery}(v) \wedge u.r_v = v.s \longrightarrow\ u.r_v := v.q$

od

$$
\begin{aligned}
\mathit{validQuery}(v) &\equiv v.q = v.s + 1 \bmod 3 \\
\mathit{dialogPaused}(v) &\equiv v.q = v.s = \textsc{Paused} \wedge \\
&\qquad \forall u \in N(v) : u.r_v = \textsc{Paused} \\
\mathit{dialogConsistent}(v) &\equiv \mathit{dialogPaused}(v) \vee (\mathit{validQuery}(v) \wedge \\
&\qquad \forall u \in N(v) : u.r_v \in \{v.s, v.q\}) \\
\mathit{dialogAcknowledged}(v) &\equiv \mathit{validQuery}(v) \wedge \forall u \in N(v) : u.r_v = v.q
\end{aligned}
$$

Figure 5.11: Implementation of the dialog within cell C_v



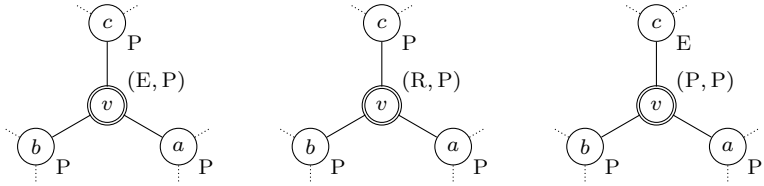
(a) dialog-paused (b) dialog-acknowledged (c) dialog-consistent

Figure 5.12: Dialog-variables of a dialog-consistent cell C_v. The label next to v shows $(v.s, v.q)$. The other labels show $a.r_v$, $b.r_v$, and $c.r_v$. P = Paused, R = Repaired, E = Executed

shown in Figure 5.10c. The general rule is as follows: For each node $v \in V$ one instance (v, Q) and an instance (v, R_u) for all $u \in N(v)$ exists. The implementation of protocols Q and R_v is shown in Figure 5.11. The implementation is incomplete but serves the purpose of explaining the dialog between each of the center instances and the responding instances. It is slightly extended in Section 5.8.3.

Instance (v, Q) uses the variable $v.s$ to store the current position of cell C_v. If $v.q$ differs from $v.s$, then the pair $(v.s, v.q)$ is a *query* for a transition from position $v.s$ to $v.q$. If $v.q$ equals $v.s$, then the pair is called a *pause*. Queries for a transition not shown in Figure 5.9 (for example from Executed to Repaired) and pauses at any position other than Paused are considered *invalid*. Such invalid values indicate an erroneous state and basically lead to a reset of the cell's dialog (explained below). An instance $(u, R_v), u \in N(v)$ acknowledges a query by setting $u.r_v := v.q$ in Rule R2. It holds that $u.r_v = v.s$ if the query has not been acknowledged yet. All variables $u.r_v$, $u \in N(v)$ are referred to as *response variables*. The variables $v.s$ and $v.q$ as well as the response variables are also called *dialog-variables*.

Consider a cell in which all dialog-variables have the value Paused, as shown in Figure 5.12a. In this state, the cell is called *dialog-paused*. The center instance (v, Q) starts a new cycle by setting the variable $v.q$ to Repaired in Rule Q2. The Boolean predicate $startCond_Q(v)$ controls whether a new cycle is started. The predicate is formally defined in Section 5.8.3. Informally, it is satisfied if the primary variables need repairing or if the node v is enabled with respect to $\mathcal{A}$. If all responding instances acknowledge the query, then C_v is called *dialog-acknowledged*. This is depicted in Figure 5.12b. The center node then completes the transition by setting



(a) dialog-consistent (b) non-dialog-consistent (c) non-dialog-consistent

Figure 5.13: Dialog-variables of a 1-faulty cell C_v. The label next to v shows $(v.s, v.q)$. The other labels show $a.r_v$, $b.r_v$, and $c.r_v$. P = PAUSED, R = REPAIRED, E = EXECUTED

$v.s := v.q$ in Rule Q3. If the new position is not PAUSED and therefore a pause is not valid, then Rule Q3 asks for a new transition straight away by incrementing $v.q$, cf. Figure 5.12c. A cycle of the state machine has been completed, when the cell has reached position PAUSED. During the whole process, a cell always stays *dialog-consistent*, i.e., the cell is either dialog-paused or $(v.s, v.q)$ is a valid query and the response variables are either equal to $v.s$ or $v.q$. The Boolean predicates in Figure 5.11 formalize the above notions.

Of course, in the initial configuration or in case of a fault, cells are in general not dialog-consistent. To make the dialog within a cell self-stabilizing, a reset of the dialog is implemented. First, the center instance (v, Q) sets its dialog-variables to PAUSED in Rule Q1. Afterwards, the responding instances reset the responding variables to PAUSED as well by Rule R1. This yields a dialog-paused and therefore dialog-consistent cell.

Note that at most one rule of protocol Q can be enabled at the same time. The same holds for protocol R_v. For example, consider the case that Rule Q3 is enabled. *dialogAcknowledged*(v) implies that C_v is dialog-consistent and that $(v.s, v.q) \neq (\text{PAUSED}, \text{PAUSED})$. Hence, Rules Q1 and Q2 are disabled. Also consider that *dialogPaused*(v) implies dialog-consistency as well.

In a legitimate configuration, all cells are dialog-paused. So in a 1-faulty configuration, either $(v.s, v.q)$ differs from $(\text{PAUSED}, \text{PAUSED})$ or one response variable $u.r_v \neq \text{PAUSED}$, $u \in N(v)$. The dialog-variables of such cells are shown in Figure 5.13. Most corruptions lead to a non-dialog-consistent cell (cf. Figures 5.13b and 5.13c). A reset of the dialog is completed within one move of either (v, Q) or (u, R_v), Rule Q1 or R1 respec-

tively. Clearly, not every corruption leads to a non-dialog-consistent cell and thus to a reset of the dialog. In fact, only corruptions that set $(v.s, v.q)$ to (EXECUTED, PAUSED) or (PAUSED, REPAIRED) result in a dialog-consistent cell. In the latter case, cell C_v starts a new cycle. In the former case, an execution of Rule Q3 yields a dialog-paused cell.

In any case, if the initial configuration is 1-faulty, then a cell C_v will perform the transition from PAUSED to REPAIRED, and thus the repair of a possibly corrupted primary variable, before any other transition. The only exception is the execution of Rule Q3 that finishes the transition from EXECUTED to PAUSED. However, this exception is shown to be insignificant for a successful repair.

5.8.3 Cell Transitions

To the dialog implementation in the previous section, we now add the implementation of the tasks specified by the edge labels in Figure 5.9. The implementation is split into two phases per transition: First, procedure $action_{R_v}(u)$ is invoked whenever a responding instance (u, R_v) acknowledges a query by (v, Q). Second, when the cell is dialog-acknowledged and a transition is completed, (v, Q) calls procedure $action_Q(v)$.

The revised implementation of protocols Q and R_v is shown in Figure 5.14. The implementation is identical to the one given in Figure 5.11, except the following changes: The calls to $action_Q$ and $action_{R_v}(u)$ have been added to Rules Q3 and R2 respectively. Also, Rule Q3 has been modified to avoid that cell C_v becomes dialog-paused even though $startCond_Q(v)$ is true, thus avoiding an extra round for the execution of Rule Q2. Furthermore, Rule R2 has been modified such that a responding instance (u, R_v) delays an acknowledgment if $v.q =$ EXECUTED but $repaired(u)$ is not satisfied. Roughly speaking, the Boolean predicate $repaired(u)$ is true if the primary variables of u are safe to be used. If this is not the case, then any execution of $\mathcal{A}$ should be avoided, as it would lead to contamination.

Figure 5.15 shows the implementation of $action_Q$ and $action_{R_v}$, which we proceed to explained in detail. Let $\sigma_{\mathcal{A}}$ denote the set of all possible local states of algorithm $\mathcal{A}$. Procedure $action_Q$ adds the variable $v.p \in \sigma_{\mathcal{A}}$ which stores the local state of node v with respect to $\mathcal{A}$ to the local state of (v, Q). It is the only primary variable of cell C_v. Furthermore, procedure $action_{R_v}$ adds two variables $u.d_v$ and $u.b_v \in \sigma_{\mathcal{A}}$ to the local state of each (u, R_v). The variable $u.d_v$ is called a *decision-variable* and $u.b_v$ is called a *backup* of $v.p$. The purpose of the decision-variables is explained in detail in Sections 5.8.4.2 and 5.8.4.3.

Protocol Q (complete implementation)

Nodes: v is the current node
Variables: $v.s, v.q \in \mathbb{Z}_3$

do
[Q1] $\neg \mathit{dialogConsistent}(v) \wedge (v.s, v.q) \neq (\textsc{Paused}, \textsc{Paused}) \longrightarrow$
$(v.s, v.q) := (\textsc{Paused}, \textsc{Paused})$
[Q2] $\square$ $\mathit{dialogPaused}(v) \wedge \mathit{startCond}_Q(v) \longrightarrow v.q := \textsc{Repaired}$
[Q3] $\square$ $\mathit{dialogAcknowledged}(v) \longrightarrow$
$\mathit{action}_Q(v)$; $v.s := v.q$
if $v.s \neq \textsc{Paused} \vee \mathit{startCond}_Q(v)$ **then**
$v.q := (v.s + 1) \bmod 3$
end
od

Protocol R_v (complete implementation)

Nodes: u is the current node, $v \in N(u)$ is the center of the cell
Variables: $u.r_v \in \mathbb{Z}_3$

do
[R1] $v.s = v.q = \textsc{Paused} \wedge u.r_v \neq \textsc{Paused} \longrightarrow u.r_v := \textsc{Paused}$
[R2] $\square$ $\mathit{validQuery}(v) \wedge u.r_v = v.s \wedge (v.q = \textsc{Executed} \Rightarrow \mathit{repaired}(u))$
$\longrightarrow \mathit{action}_{R_v}(u)$; $u.r_v := v.q$
od

$$\mathit{backupConsistent}(v) \equiv \forall u \in N(v) : u.b_v = v.p$$
$$\mathit{repaired}(v) \equiv \mathit{dialogConsistent}(v) \wedge (\mathit{backupConsistent}(v) \vee v.s = \textsc{Repaired} \vee (v.s = \textsc{Executed}) \wedge (\forall u \in N(v) : u.r_v = \textsc{Paused} \Rightarrow u.b_v = v.p)))$$
$$\mathit{startCond}_Q(v) \equiv G_{\mathcal{A}}(v) \vee \neg \mathit{backupConsistent}(v)$$

The predicates *dialogPaused*, *dialogConsistent*, *dialogAcknowledged*, and *validQuery* are as defined in Figure 5.11.

Figure 5.14: Full implementation of protocols Q and R_v

Procedure $action_Q(v)$

Nodes: v is the current node
Variables: $v.p \in \sigma_{\mathcal{A}}$

if $v.q = \textsc{Repaired}$ **then**
  $u :=$ any neighbor of node v
  if $backupConsistent(v)$ **then** Keep value of $v.p$
  else if $\deg(v) > 1 \wedge \forall w \in N(v) : w.b_v = u.b_v$ **then** $v.p := u.b_v$
  else if $\deg(v) = 1 \wedge u.d_v = \textsc{Update}$ **then** $v.p := u.b_v$
  else if $\deg(v) = 1 \wedge u.d_v = \textsc{Single}$ **then** $pairRepair(v, u)$
end
if $v.q = \textsc{Executed}$ **then**
  if $G_{\mathcal{A}}$ **then** $S_{\mathcal{A}}$
end

Procedure $pairRepair(v, u)$

if $v.b_u \neq u.p \wedge v.id < u.id \wedge$
  $\neg pairEnabled_{\mathcal{A}}(v : v.p, u : v.b_u)$ **then** Keep value of $v.p$
else if $pairEnabled_{\mathcal{A}}(u : u.p, v : v.p) \wedge$
  $\neg pairEnabled_{\mathcal{A}}(u : u.p, v : u.b_v)$ **then** $v.p := u.b_v$

Procedure $action_{R_v}(u)$

Nodes: u is the current node, $v \in N(u)$ is the center of the cell
Variables: $u.r_v \in \{\textsc{Keep}, \textsc{Update}, \textsc{Single}\}; u.b_v \in \sigma_{\mathcal{A}}$

if $v.q = \textsc{Repaired}$ **then**
  if $\deg(u) = 1$ **then** $u.d_v := \textsc{Single}$
  else if $(\forall w \in N(u) : w = v \vee w.b_u = u.p) \wedge$
    $(G_{\mathcal{A}}(u) \vee G_{\mathcal{A}}(v : v.p, u : u.p))$ **then** $u.d_v := \textsc{Update}$
  else $u.d_v := \textsc{Keep}$
end
if $v.q = \textsc{Paused}$ **then**
  $u.b_v := v.p$
end

Figure 5.15: Implementation of transition actions

A *cycle* starts with a transition from PAUSED to REPAIRED. This transition, which repairs a corrupted primary variable, is explained in detail in Section 5.8.4. But in short, procedure $action_{R_v}$ first provides information through the decision variables used by $action_Q$ to restore the value of the primary variable of the cell to its value prior to the corruption. Note that this repair is only attempted once, i.e., in any subsequent cycle, the primary variable is not modified. During the transition to EXECUTED, a single move of algorithm $\mathcal{A}$ is executed by $action_Q$. Without loss of generality $\mathcal{A}$ is assumed to have been rewritten such that it uses the variables $v.p$ on each node $v \in V$. Procedure $action_{R_v}$ does not perform any task during that particular transition. A final transition back to PAUSED completes each cycle. During this transition, the query by the center node is thought of as a signal for the responding instances to update their backup copies. This is implemented in $action_{R_v}$. Procedure $action_Q$ does not perform any task during this transition.

After completing a cycle, all backups coincide with the primary variable of the cell. In this state, the cell is called *backup-consistent*. But during convergence, the backup-consistency of a cell is destroyed regularly, for example if a move of $\mathcal{A}$ has just been executed, but the backup copies have not been updated yet. Hence, backup-consistency cannot be used as an indicator, whether a fault has already been repaired or not. A new predicate *repaired*(v) is defined: A cell is called *repaired* if it is dialog-consistent and

- backup-consistent or
- at position REPAIRED or EXECUTED.

These are the two positions at which a cell is expected to be non-backup-consistent. In addition, every backup of a responding instance that has acknowledged the transition to PAUSED is required to be consistent. Note that the position REPAIRED just indicates that a cell has tried to repair its primary variable while *repaired*(v) indicates a kind of consistency which means that cell C_v does not attempt any further repair of the primary variable.

It remains to explain the conditions under which a dialog-paused cell starts a new cycle. Certainly, a cycle should be started if the cell is not repaired. Also, cell C_v should be started if node v is enabled with respect to $\mathcal{A}$, so that the primary configuration eventually converges. The definition of $startCond_Q(v)$ as given in Figure 5.14 reflects this. Note that $startCond_Q(v)$ is only evaluated if $v.s =$ PAUSED, in which case *repaired*(v) is equivalent to *backupConsistent*(v).

Recall the two objectives defined at the beginning of Section 5.8 and that the first transition executed by any 1-faulty cell is the one from PAUSED to REPAIRED. The latter is clearly also true for a dialog-paused cell. $action_{R_v}$ can thus provide the necessary 2-hop knowledge that $action_Q(v)$ depends on in order to properly repair $v.p$. Furthermore, delaying acknowledgments keeps (v, Q) from executing moves of $\mathcal{A}$ until all neighboring cells are repaired. In effect contamination is prevented. Hence, both objectives are achieved.

It is worth mentioning that the local synchronization implemented by delaying acknowledgments bears the potential for deadlocks. If the progress of a cell's dialog depends on such a delayed acknowledgment, then the cell is called *blocked*. The following predicate formalizes this:

$$\begin{aligned} \mathit{blocked}(v) \equiv \mathit{dialogConsistent}(v) \wedge v.q = \textsc{Executed} \wedge \\ \exists u \in N(v) : u.r_v \neq v.q \wedge \neg \mathit{repaired}(u) \end{aligned}$$

If C_u does not become repaired eventually, C_v remains blocked forever. It is easy to see that any blocked cell is in fact repaired, as $\mathit{blocked}(v)$ implies dialog-consistency of C_v and $v.s = \textsc{Repaired}$. Hence $\mathit{repaired}(v)$ is satisfied and no circular waits between blocked cells can occur. Lemma 5.19 shows that any non-repaired cell is enabled and becomes repaired eventually. Furthermore, Lemma 5.21 shows that $\mathit{repaired}(v)$ is stable for any execution of $\mathcal{A}_{FL}$, i.e., once a cell is repaired, it remains repaired thereafter. Hence, blocked cells only exist in an initial phase. Once all cells are repaired, cells are completely independent in the sense that no cell can become blocked.

5.8.4 Fault-Repair

A cell C_v is called *legitimate* if it is dialog-paused and backup-consistent. A configuration is defined to be *legitimate* if all cells are legitimate and all nodes are disabled with respect to $\mathcal{A}$. Indeed, Section 5.8.5.1 shows that the distributed algorithm $\mathcal{A}_{FL}$ with $\mathcal{A}_{FL}(v) = \{Q\} \cup \{R_u \mid u \in N(v)\}$ for all $v \in V$ is self-stabilizing with respect to the following predicate:

$$\mathcal{L}_{\mathcal{A}_{FL}} \equiv \forall v \in V : \mathit{dialogPaused}(v) \wedge \neg \mathit{startCond}_Q(v)$$

where $\neg \mathit{startCond}_Q(v)$ implies that all cells are backup-consistent and all nodes are disabled with respect to $\mathcal{A}$. A 1-faulty configuration differs from a legitimate configuration in the variables of a single node only. It follows that in a 1-faulty configuration, cells are either legitimate or differ from a dialog-paused and backup-consistent cell in the variables of a single node

only. Such a cell is called 1-*faulty*. Notice that there can be at most $\Delta + 1$ non-legitimate cells in a 1-faulty configuration.

Assume that cell C_v is 1-faulty. Recall from Section 5.8.2 that C_v becomes dialog-consistent at position PAUSED within a few moves and starts a new cycle if $startCond_Q(v)$ is satisfied. The transition from PAUSED to REPAIRED is the first transition executed by C_v. The center instance (v, Q) may execute Rule Q3 beforehand. In that case it holds that $v.q = \text{PAUSED}$ and the invocation of $action_Q(v)$ does not change the primary variable of C_v. So when the transition from PAUSED to REPAIRED is performed by C_v, the backups and the primary variable have not been modified yet and their values are still equal to the ones in the 1-faulty initial configuration.

This section discusses the case where C_v is not backup-consistent and how it detects and repairs a corrupted primary variable. Assume that (v, Q) has made the query (PAUSED, REPAIRED). All responding instances call $action_{R_v}$ and set their decision-variables. Afterwards, the center instance calls $action_Q$ and completes the transition. The repair of the primary variable of C_v happens during this handshake between $action_{R_v}$ and $action_Q$. The repair mechanism is inspired by the synchronous mechanism described by Ghosh et al. [GGHP07]. It was refined and optimized so that it fits into a single transition of the state-machine. Each of the following three subsections describes one of three special cases that need to be considered during a repair. The repair is always explained from the perspective of cell C_v.

5.8.4.1 Multiple backups

If node v has multiple neighbors, then there are multiple backups in cell C_v and it is easy to decide whether the primary variable has been corrupted. Prior to the fault, C_v has been backup-consistent. Either, a backup or the primary variable has been corrupted by the fault. Therefore, one of the two following cases must match:

a) The value of $v.p$ differs from the backups and all backups in cell C_v are identical.

b) There is one backup that differs from the value of $v.p$. All other backup copies are equal to the value of $v.p$.

Case a) is identified simply by checking, whether all backups have the same value. If this is the case, then $v.p$ must be the variable which has been corrupted and $v.p$ is updated with the value of the backups in line 4 of $action_Q(v)$. Otherwise, it can be assumed that a backup has been corrupted.

5.8.4.2 Leaf Nodes: Only a Single Backup

Let v denote a node with only one neighbor u. Then C_v is a cell with only one responding instance and thus only one backup. It is assumed that u has more than one neighbor. The case where u has only one neighbor is discussed separately in Section 5.8.4.3.

Obviously the method explained in Section 5.8.4.1 does not apply. It relies on the fact that the corrupted value is outnumbered by the correct value. But in this case, only one backup is available. If C_v is not backup-consistent, then either $v.p$ or $u.b_v$ are corrupted. The basic strategy is to let node u decide. Node u will pass its decision to v in the variable $u.d_v$. The following two cases explain how $u.d_v$ is set.

a) Assume that $v.p$ and possibly $v.b_u$ have been corrupted. Cell C_u is backup-consistent, with the exception of $v.b_u$. Furthermore it is assumed that at least one of v or u is enabled with respect to $\mathcal{A}$. Otherwise, the corruption of $v.p$ is considered to be harmless and $v.p$ does not need to be changed. The check in line 3 of $action_{R_v}(u)$ evaluates to true and $u.d_v$ is set UPDATE accordingly.

b) Assume that $u.b_v$ and possibly $u.p$ have been corrupted. If $u.p$ has been corrupted, $u.p$ differs from all backups in cell C_u. Hence, the condition in line 3 of $action_{R_v}(u)$ is false. If $u.p$ has not been corrupted, then cell C_u is backup-consistent, but both v and u are disabled with respect to $\mathcal{A}$. The condition in line 3 is false and $action_{R_v}(u)$ sets its decision-variable to KEEP.

Note that it is not a problem for node u to evaluate $G_{\mathcal{A}}(v)$. If u is the only neighbor of v, then $G_{\mathcal{A}}(v)$ depends on $v.p$ and $u.p$ only. Also note that node u cannot know whether v has more than one neighbor or not. This does however not pose a problem, as the value of $u.d_v$ is only used by $action_Q(v)$ if v has exactly one neighbor. Hence, $action_{R_v}(u)$ simply assumes that $\deg(v) = 1$. This avoids an additional handshake between v and u and additional corruptible variables.

While C_v executes its first cycle, cell C_u may do the same. In fact, depending on the choices of the scheduler, C_u may be faster than C_v. As a result, cell C_u repairs its primary variable before v can do so. Cell C_u may restore the original value of $u.p$ if it detects a corruption of $u.p$. From the perspective of cell C_v, it will simply seem as if $u.p$ has never been corrupted. This is already covered by the assumptions in case b). Furthermore, cell C_u cannot progress to position EXECUTED as long as C_v is not repaired

as C_u becomes blocked. This suffices to ensure that the check in line 3 of $action_{R_v}(u)$ yields the correct result as long as needed.

5.8.4.3 The Single-Edge Case

The system may consist of only two adjacent nodes. This case needs special treatment to successfully repair a fault. The methods from Sections 5.8.4.1 and 5.8.4.2 do not work for this special topology.

Let v and u denote the two neighboring nodes. During transition from PAUSED to REPAIRED, both $action_{R_v}(u)$ and $action_{R_u}(v)$ set the decision-variables $v.d_v$ and $u.d_v$ to SINGLE. This allows a reliable detection of this topology in $action_Q(v)$ and $action_Q(u)$ in line 6. If this topology is detected, they invoke $pairRepair(v, u)$ and $pairRepair(u, v)$, which perform the actual repair of either $v.p$ or $u.p$ respectively.

Assume that v is the faulty node, i.e., either $v.p$ or $v.b_u$ is corrupted. It holds that C_v or C_u is not backup-consistent. If both cells are not backup-consistent, then the configuration may be symmetric in the sense that both the assignments $v.p := u.b_v$ or $u.p := v.b_u$ can be used to obtain a legitimate primary configuration. In this case, the node identifiers are used for symmetry breaking in order to avoid that simultaneous execution of $pairRepair(v, u)$ and $pairRepair(u, v)$ erroneously overrides both primary variables with the value of a backup. The primary variable of the node with the larger identifier will be overwritten with the backup on the node with the lower identifier. The case where only one cell is not backup-consistent is not symmetrical and is resolved without node identifiers.

a) Assume that only $v.p$ has been corrupted. Note that the condition in line 1 of $pairRepair(v, u)$ is false since $v.b_u = u.p$. If both nodes are disabled with respect to $\mathcal{A}$ even though $v.p$ has been corrupted, then the condition in line 2 is also false and the corruption is considered to be harmless and is ignored. Otherwise, if one of the two nodes is enabled with respect to $\mathcal{A}$, then $v.p$ is updated with $u.b_v$ in line 2 of $pairRepair(v, u)$. As C_u is backup-consistent, the condition in line 3 is true and $action_Q(u)$ does not call $pairRepair(u, v)$.

b) Assume that only $v.b_u$ has been corrupted. Since C_v is still backup-consistent, $pairRepair(v, u)$ is never invoked by $action_Q(v)$. When $pairRepair(u, v)$ is invoked by $action_Q(u)$, the condition in line 1 is false since $u.b_v = v.p$. The condition in line 2 is also false since both nodes are still disabled with respect to $\mathcal{A}$. Hence, $u.p$ is not changed.

c) Assume that $v.p$ and $v.b_u$ have been corrupted and that $v.id > u.id$. If $pairRepair(u, v)$ is invoked before or simultaneous to $pairRepair(v, u)$, then the condition in line 1 of $pairRepair(u, v)$ is true and $u.p$ is not modified. When $pairRepair(v, u)$ is invoked, the condition in line 1 is false since $v.id > u.id$ and $v.p$ is set to $u.v_c$ in line line 2. The corresponding condition in line 2 is true since $u.b_v$ still holds the uncorrupted value of $v.p$. Note that $v.p$ is only overwritten if v or u are actually enabled with respect to $\mathcal{A}$. Otherwise, the corruption of $v.p$ can be ignored. Now that it holds $v.p = u.b_v$ and case b) applies and explains why $pairRepair(u, v)$ leaves $u.p$ unchanged if invoked subsequent to $pairRepair(v, u)$.

d) Assume that $v.p$ and $v.b_u$ have been corrupted and that $v.id < u.id$. In the symmetrical case, $u.p$ and $u.b_v$ can be seen as the corrupted values, and case c) applies and explains why $u.p$ is updated with the value of $v.b_u$ and why $v.p$ is left unchanged.

   Otherwise, in the asymmetric case, any invocation of $pairRepair(u, v)$ prior to or simultaneously with $pairRepair(v, u)$ leaves $u.p$ unchanged. The condition in line 2 is false since the $pairEnabled_{\mathcal{A}}$-based check reveals that setting $u.p$ to $v.b_u$ does not disable both nodes with respect to $\mathcal{A}$. When $pairRepair(v, u)$ is invoked, the condition in line 1 is false since setting $u.p$ to $v.b_u$ would not lead to a successful repair. The condition in line 2 is true and $v.p$ is set to $u.v_c$, unless nodes v and u are already disabled with respect to $\mathcal{A}$. Now that it holds $v.p = u.b_v$, case b) applies and explains why $pairRepair(u, v)$ leaves $u.p$ unchanged if invoked subsequently to $pairRepair(v, u)$.

In all cases, at most one of the two nodes updates its primary variable. Afterwards, both nodes are disabled with respect to $\mathcal{A}$ and the primary configuration is legitimate again.

5.8.5 Analysis

Recall the definition of legitimate cells and $\mathcal{L}_{\mathcal{A}_{FL}}$ at the beginning of Section 5.8.4. First it is shown that algorithm $\mathcal{A}_{FL}$ is silent and self-stabilizing with respect to $\mathcal{L}_{\mathcal{A}_{FL}}$. Second, it is shown that $\mathcal{A}_{FL}$ is fault-containing with respect to $\mathcal{L}_{\mathcal{A}}$. In addition, an analysis of the fault-gap and the stabilization time is given. The complexity analysis is performed by counting rounds. However, the definition of convergence requires showing that the algorithm reaches a legitimate configuration after a finite number of steps. Hence, it

does not suffice to prove that the number of rounds is finite. Note that as discussed in Section 2.4, a round under an unfair scheduler may consist of an infinite number of steps unless it is shown that any execution of $\mathcal{A}_{FL}$ is of finite length. Therefore, Theorem 5.26 and the related Lemmas 5.12 and 5.19 also count moves.

The proofs are mainly based on observations about the state and behavior of individual cells. Recall that the state and behavior of a cell C_v is determined by instance (v, Q) and all instances (u, R_v), $u \in N(v)$. Correspondingly, a cell is called *enabled* if at least one of its instances is enabled and it is said that cell C_v makes a *move* if one of its instances makes a move. As shown by Lemma 5.14, the dialog within each cell makes it particularly easy to analyze the round-complexity of $\mathcal{A}_{FL}$.

The following theorems summarize the main results:

Theorem 5.8. *The transformation is correct, i.e., $\mathcal{L}_{\mathcal{A}_{FL}}$ implies $\mathcal{L}_{\mathcal{A}}$ for all configurations.*

Proof. Let c denote configuration that satisfies $\mathcal{L}_{\mathcal{A}_{FL}}$. All nodes are disabled with respect to $\mathcal{A}$ in c. Since $\mathcal{A}$ is self-stabilizing, c satisfies $\mathcal{L}_{\mathcal{A}}$. □

Theorem 5.9 (Stabilization Space)**.** *The space required per node by $\mathcal{A}_{FL}$ in a legitimate configuration is $\mathcal{O}(\Delta s_{\mathcal{A}})$ where $s_{\mathcal{A}}$ is the space required by $\mathcal{A}$ per node in a legitimate configuration.*

Proof. Each nodes stores up to Δ backups. The backups and the primary variable account for $\mathcal{O}(\Delta s_{\mathcal{A}})$ bits. The dialog and decision variables take $\mathcal{O}(\Delta)$ bits per node. □

Theorem 5.10 (Self-Stabilization)**.** *$\mathcal{A}_{FL}$ is silent and self-stabilizing with respect to $\mathcal{L}_{\mathcal{A}_{FL}}$. $\mathcal{A}_{FL}$ terminates after at most $6T + 11$ rounds, where T denotes the maximum length of any execution of $\mathcal{A}$ in rounds.*

Theorem 5.11 (Fault-Containment)**.** *$\mathcal{A}_{FL}$ is fault-containing with respect to $\mathcal{L}_{\mathcal{A}}$ with a containment time and fault-gap of $\mathcal{O}(1)$ rounds, a contamination number of 1, and a fault-impact of $\mathcal{O}(\Delta^2)$ (radius 2).*

Sections 5.8.5.1 and 5.8.5.2 prove the correctness of Theorems 5.10 and 5.11 respectively. The proofs assume that per step at most one instance per cell is selected. This greatly simplifies the proof. This assumption is clearly true for the central scheduler. In Section 5.8.5.3 it is shown that a partial serialization for any step of $\mathcal{A}_{FL}$ under the distributed scheduler exists, such that in each step of the serialization at most one instance per cell makes a move.

5.8.5.1 Self-Stabilization

Lemma 5.12. *If a cell is not dialog-consistent, then it is enabled and becomes dialog-paused (and hence dialog-consistent) after at most* 2Δ *moves. These moves happen within the next* 2 *rounds.*

Proof. Let C_v denote a cell that is not dialog-consistent.

Case a) $v.s = v.q = \textsc{Paused}$. Since v is not dialog-consistent, at least one response-variable is not equal to PAUSED. Within one round, Rule R1 of R_v resets all responding variables them to PAUSED. Note that all rules of (v, Q) are disabled until that happens.

Case b) $\mathit{validQuery}(v)$. Rule Q1 of (v, Q) is enabled since there is at least one $u.r_v$, $u \in N(v)$ that violates dialog-consistency. (u, R_v) is disabled. Depending on the value of the other response-variables, some responding instances of C_v may execute Rule R2 before Rule Q1 of (v, Q) resets $v.s$ and $v.q$ to PAUSED. This happens within one round. Now case a) applies.

Case c) $\neg(\mathit{validQuery}(v) \vee v.s = v.q = \textsc{Paused})$. All responding instances of C_v are disabled. Rule Q1 of (v, Q) sets $v.s$ and $v.q$ to PAUSED within 1 round. If C_v is not yet dialog-paused (and hence dialog-consistent), case a) applies. □

Lemma 5.13. *A dialog-consistent cell* C_v *remains dialog-consistent under the execution of* $\mathcal{A}_{FL}$.

Proof. Let C_v denote a dialog-consistent cell and let (u, R_v), $u \in N(v)$ denote a responding instance. Rule Q1 of (v, Q) as well as Rule R1 of (u, R_v) are disabled since C_v is dialog-consistent.

If Rule Q2 of (v, Q) is enabled, then C_v is dialog-paused. The rule set $v.q :=$ REPAIRED. Since all response variables are equal to PAUSED, dialog-consistency holds after executing Rule Q2.

If Rule Q3 of (v, Q) is enabled, then all response variables are equal to $v.q$ for all $w \in N(v)$. After execution of Rule Q3, they are all equal to $v.s$. Furthermore, Rule Q3 ensures that $(v.s, v.q)$ is a valid query or pause. Hence C_v remains dialog-consistent.

If Rule R2 of (u, R_v) is enabled, then $(v.s, v.q)$ is a valid query. C_v remains dialog-consistent when changing $u.r_v$ from $v.s$ to $v.q$. □

For a dialog-consistent cell C_v, it is said that it starts a new cycle if (v, Q) sets $(v.s, v.q) :=$ (PAUSED, REPAIRED). This happens either via Rule Q2 or Rule Q3. Cell C_v finishes a cycle if C_v completes the transition to position PAUSED. A dialog-consistent cell C_v completes the transition to position x

with the move that sets $v.s := x$. This is always a result of an execution of Rule Q3 of (v, Q). Note that one move of (v, Q) can both finish a cycle and start a new cycle provided that $startCond_Q(v)$ is true.

Lemma 5.14. *Let C_v denote a dialog-consistent cell that is not dialog-paused. The transition from the current position of C_v to the subsequent position takes at most $\Delta + 1$ moves. These moves happen within at most 2 rounds unless the cell becomes blocked.*

Proof. Assume that $(v.s, v.q)$ is a valid query and that none of the responding instances have acknowledged the query of the center node yet. (v, Q) is disabled until all responding nodes acknowledge the current query.

All responding instances are enabled with respect to Rule R2 and thus acknowledge the query by (v, Q) within one round. (v, Q) is now enabled with respect to Rule Q3 and completes the transition within at most one more round. □

Corollary 5.15. *A dialog-consistent non-dialog-paused cell is enabled, unless it is blocked.*

Corollary 5.16. *A dialog-consistent non-dialog-paused cell at position x finishes its current cycle with at most $(3 - x)(\Delta + 1)$ moves. These moves happen within at most $2(3 - x)$ rounds, unless $x <$ EXECUTED and the cell becomes blocked.*

Corollary 5.17. *A dialog-paused cell starts a cycle and reaches position x with at most $1 + x(\Delta + 1)$ moves. These moves happen within at most $2x + 1$ rounds, unless $x \geq$ EXECUTED and the cell becomes blocked. The cell finishes the cycle after at most $1 + 3(\Delta + 1)$ moves. These moves happen within 7 rounds, unless the cell becomes blocked.*

Corollary 5.18. *A dialog-consistent cell C_v at position* EXECUTED *finishes its current cycle, starts a new cycle, and reaches position* EXECUTED *within at most 6 rounds, assuming that C_v does not become blocked and that $startCond_Q(v)$ is satisfied continuously.*

Note that it takes a cell C_v only 6 rounds for a cycle from EXECUTED to EXECUTED, as the optimization of Rule Q3 allows C_v to avoid becoming dialog-paused unnecessarily if $startCond_Q(v)$ is satisfied.

Lemma 5.19. *If a cell is not repaired, then it is enabled and becomes repaired after at most $3\Delta + 3$ moves. These moves happen within the next 5 rounds.*

Proof. Let C_v be a non-repaired cell.

Case a) v is not dialog-consistent in c_0. By Lemma 5.12 v becomes dialog-paused after 2Δ which happen within at most 2 rounds. Now case b) applies.

Case b) C_v is dialog-consistent with $v.s = \text{PAUSED}$. By Corollary 5.17, the cell reaches position REPAIRED with $\Delta + 2$ moves which happen within at most 3 rounds.

Case c) C_v is dialog-consistent and $v.s \neq \text{PAUSED}$. If $v.s = \text{REPAIRED}$, then C_v is repaired. Otherwise, $v.s = \text{EXECUTED}$. In order for C_v not to be repaired, a responding instance (u, R_v), $u \in N(v)$ with $u.r_v = \text{PAUSED}$ and $u.b_v \neq v.p$ must exist. By Corollary 5.16, the cell completes the current cycle after at most $\Delta + 1$ moves which happens within at most 2 rounds. Then it holds $v.s = \text{PAUSED}$ and case b) applies.

Note that C_v remains dialog-consistent under any moves in cases b) and c) by Lemma 5.13. Also note that C_v never attempts to performs a transition to EXECUTED and thus cannot become blocked. □

Corollary 5.20. *Let C_v denote a non-repaired cell. The move that renders C_v repaired is the first modification of $v.p$.*

The proof follows from the proof of Lemma 5.19.

Lemma 5.21. *A repaired cell C_v remains repaired under the execution of $\mathcal{A}_{FL}$.*

Proof. Let C_v denote a repaired cell. Then C_v is also dialog-consistent. By Lemma 5.13, it remains dialog-consistent under the execution of $\mathcal{A}_{FL}$.

Case a) $(v.s, v.q) = (\text{PAUSED}, \text{PAUSED})$. It follows that C_v is backup-consistent since $v.s = \text{PAUSED}$. Rule Q2 of (v, Q) does not change $v.p$. Hence, C_v remains backup-consistent and thus repaired. No other rules are enabled within C_v.

Case b) $(v.s, v.q) = (\text{PAUSED}, \text{REPAIRED})$. It follows that C_v is backup-consistent since $v.s = \text{PAUSED}$. Rule R2 of (u, R_v), $u \in N(v)$ does not change $u.b_v$. Hence, C_v remains backup-consistent and thus repaired. Rule Q3 of (v, Q) may change $v.p$. However, the same move sets $v.s :=$ REPAIRED and thus C_v remains repaired. Note that all responding variables have the value REPAIRED in this case.

Case c) $(v.s, v.q) = (\text{REPAIRED}, \text{EXECUTED})$. C_v might not be backup-consistent. C_v is repaired solely since $v.s = \text{REPAIRED}$. Hence, Rule R2 of any responding instance does not affect the value of *repaired*(v). Rule Q3 of (v, Q) may change $v.p$. However, the same move sets $v.s :=$ EXECUTED

and thus C_v remains repaired. Note that all responding variables have the value EXECUTED in this case.

Case d) $(v.s, v.q) = (\text{EXECUTED}, \text{PAUSED})$. Since C_v is repaired it holds that $u.b_v = v.p$ for all $u \in N(v)$ with $u.r_v = v.q$. Rule R2 of (u, R_v), $u \in N(v)$ also sets $u.b_v := v.p$ if it sets $u.r_v := v.q$. Hence, C_v remains repaired. When Rule Q3 of (v, Q) is enabled it holds that sets $v.s :=$ PAUSED. it holds that $u.b_v = v.p$ for all responding instances $u \in N(v)$. Hence, C_v is backup consistent. Rule Q3 of (v, Q) does not change $v.p$, hence C_v remains backup-consistent and thus repaired. □

Lemma 5.22. *For a repaired cell C_v, any modification of $v.p$ is the result of moves of $\mathcal{A}$.*

Proof. Let C_v denote a repaired and thus dialog-consistent cell. The only modification of $v.p$ that is not the result of moves of $\mathcal{A}$ happens in $action_Q(v)$ if $(v.s, v.q) = (\text{PAUSED}, \text{REPAIRED})$. Since $v.s = \text{PAUSED}$, C_v being repaired implies that C_v is backup-consistent. When called by Rule Q3 of (v, Q), the condition in line 3 ensures that $action_Q(v)$ does not modify $v.p$. □

Lemma 5.23. *A repaired cell C_v starts a new cycle only if $G_{\mathcal{A}}(v)$ is satisfied.*

Proof. Since C_v is repaired, it is also dialog-consistent. A new cycle is started either via Rule Q2 or Q3 of (v, Q).

Case a) $v.s = v.q = \text{PAUSED}$. Since C_v is repaired, it follows that C_v is backup-consistent. Hence, for Rule Q2 to be enabled, $startCond_Q(v)$ must be satisfied which requires $G_{\mathcal{A}}(v)$.

Case b) $(v.s, v.q) = (\text{EXECUTED}, \text{PAUSED})$. Because C_v is repaired and dialog-acknowledged, it holds $u.r_v = \text{PAUSED}$ and thus $u.b_v = v.p$ for all $u \in N(v)$. Hence, C_v is backup-consistent and $startCond_Q(v)$ is true if and only if $G_{\mathcal{A}}(v)$ is satisfied. Therefore, Rule Q3 starts a new cycle by setting $v.q :=$ REPAIRED only if $G_{\mathcal{A}}(v)$ is satisfied. □

Theorem 5.24 (Partial Correctness)**.** *If all nodes are disabled with respect to algorithm $\mathcal{A}_{FL}$, then $\mathcal{L}_{\mathcal{A}_{FL}}$ is satisfied.*

Proof. From Lemmas 5.12 and 5.19 it follows that all cells are dialog-consistent and repaired. Hence, no cell can be blocked. From Corollary 5.15 it follows that all cells are dialog-paused. From Rule Q2 being disabled for all (v, Q) it follows that $\neg startCond_Q(v)$ and thus $\neg G_{\mathcal{A}}(v)$ for all $v \in V$. Hence, $\mathcal{L}_{\mathcal{A}_{FL}}$ is satisfied. □

Theorem 5.25 (Closure)**.** *If $\mathcal{L}_{\mathcal{A}_{FL}}$ is true, then all nodes are disabled with respect to algorithm $\mathcal{A}_{FL}$.*

Proof. If $\mathcal{L}_{\mathcal{A}_{FL}}$ is true, then all cells are dialog-paused and backup-consistent. Furthermore, all nodes are disabled with respect to $\mathcal{A}$. All responding instances of a cell C_v and Rules Q1 and Q3 of (v, Q) are disabled since $\mathit{dialogPaused}(v)$. For all $v \in V$ the predicate $\mathit{startCond}_Q(v)$ is false since $\mathit{backupConsistent}(v)$ and $\neg G_{\mathcal{A}}(v)$. Thus Rule Q2 is disabled for all center instances. □

Theorem 5.26 (Termination)**.** *Algorithm $\mathcal{A}_{FL}$ terminates after a finite number of moves.*

Proof. In the remainder of this proof, a modification of a cell's primary variable is called a *primary change*. Primary changes happen on exactly two occasions: the move of (v, Q) that completes the transition to Repaired and the moves of (v, Q) that completes the transition to Executed. First, it is shown that there is an upper bound on the number of primary changes in the system. Second, it is shown that for each cycle of a cell there is at least one primary change in the system.

As long as there are no primary changes due to a repair, the number of possible moves of algorithm $\mathcal{A}$ is finite since algorithm $\mathcal{A}$ is silent. After each primary change that is the result of a repair, the stabilization process of algorithm $\mathcal{A}$ may start over. However, this happens at most n times, since by Lemmas 5.21 and 5.22 and Corollary 5.20 each cell C_v attempts at most one repair of $v.p$. In conclusion, the total number of primary changes in the system is finite.

Let C_v denote a cell. By Lemmas 5.12 and 5.19, C_v is dialog-consistent and repaired after a finite number of moves. By Corollary 5.16, C_v finishes the current cycle within a finite number of moves. Afterwards, C_v only starts another cycle if $G_{\mathcal{A}}(v)$ is satisfied by Lemma 5.23. By the time (v, Q) is about to finish the transition to Executed, algorithm $\mathcal{A}$ might be disabled on node v. Then a primary change must have been performed by a cell neighboring C_v. Otherwise, (v, Q) executes a move by $\mathcal{A}$ and causes a primary change.

In conclusion, at least one primary change happens during each cycle of a cell (either in the cell itself, or in a neighboring cell). Since the number of primary changes in the system is finite, the number of cycles per cell must also be finite. Hence, the total number of moves is finite. □

Lemma 5.27 (Slowdown)**.** *Let $e = \langle c_0, c_1, c_2, \ldots \rangle$ denote an execution of $\mathcal{A}_{FL}$ and let c_i denote the first configuration in which all cells are dialog-consistent and repaired. If any execution of algorithm $\mathcal{A}$ starting in c_i terminates within at most T rounds, then the length of e is at most $6T + 11$ rounds.*

Proof. By Lemmas 5.12, 5.13, 5.19 and 5.21, it takes at most 5 rounds to reach c_i and all cells remain dialog-consistent and repaired in all configurations subsequent to c_i. By Lemma 5.22, all modifications of a primary variable subsequent to c_i are the result of an execution of $\mathcal{A}$. Hence, it remains to show that every 6 rounds of $\mathcal{A}_{FL}$ subsequent to c_i are equivalent to at least one round of $\mathcal{A}$ and that all cells terminate within at most 6 rounds after $\mathcal{A}$ has terminated.

Let $e' = \langle c_j, c_{j+1}, \ldots$ with $j \geq i$ denote the subsequence of e of exactly 6 rounds. The execution e' is equivalent to one round of $\mathcal{A}$ if every node $v \in V$ either executes a move of $\mathcal{A}$ within e' or is disabled with respect to $\mathcal{A}$ in at least one configuration of e'. Let $v \in V$ denote a node that is enabled with respect to $\mathcal{A}$ in all configurations of e'. It is shown that v executed a move of $\mathcal{A}$ in e. Without loss of generality, let $v.s = \textsc{Executed}$ in c_j. By Corollary 5.18, C_v finishes its current cycle and starts a new cycle, and advances to Executed within 6 rounds. Note that C_v cannot become blocked as all cells are repaired in every configuration of e'. During the transition to Executed, C_v executes a move of $\mathcal{A}$ for node v.

When the configuration c_k, $k \geq i$ has been reached, in which all nodes are disabled with respect to $\mathcal{A}$, every cell that has already started a cycle finishes it within at most 6 rounds by Corollary 5.16. Thus, the length of e is at most $5 + 6T + 6$ rounds. □

Proof of Theorem 5.10. Convergence follows from Theorems 5.24 and 5.26. Closure is shown in Theorem 5.25. The stabilization time follows from Lemma 5.27. □

5.8.5.2 Fault-Containment

Recall that $\mathcal{L}_{\mathcal{A}}$ is the Boolean predicate that reflects whether the primary configuration is legitimate. Consider the Boolean predicate $\mathcal{L}^*_{\mathcal{A}} \equiv \forall v \in V : \neg G_{\mathcal{A}}(v)$. The predicate is true if and only if algorithm $\mathcal{A}$ has terminated.

In this section, only executions $e = \langle c_0, c_1, c_2, \ldots \rangle$ are considered that start in a 1-faulty configuration c_0. Furthermore, let $c_{\mathcal{L}}$ denote a legitimate configuration and v_f a node, such that c_0 can be derived from $c_{\mathcal{L}}$ by perturbing variables of node v_f only. There may exist multiple $c_{\mathcal{L}}$ and v_f that

c_0 can be derived from. The proofs hold for all of them. A dialog-consistent cell C_v is called *unacknowledged* if $u.r_v = v.s$ for all $u \in N(v)$. In particular, a dialog-paused cell is also unacknowledged.

To avoid an unnecessary complication of the proofs, the case where the system consists of a single edge only is not considered. A discussion of this case is given in Section 5.8.4.3.

Lemma 5.28. *All cells C_u, $u \in N[v_f]$ become dialog-consistent and unacknowledged at position* PAUSED *within at most* 1 *round. Neither the primary variable nor backups within C_u are modified.*

Proof. Case a) $u \notin N[v_f]$. None of the dialog-variables of C_u have been corrupted. Thus C_u is dialog-paused in c_0 and the claim clearly holds.

Case b) $u \in N(v_f)$. $v_f.r_u$ may have been corrupted. In this case, (v_f, R_u) resets $v_f.r_u$ to PAUSED within one round via Rule R1. This move does not modify a backup.

Case c) $u = v_f \wedge (u.s, u.q) = (\text{EXECUTED}, \text{PAUSED})$. C_u is dialog-consistent since $(v.s, v.q)$ is a valid query and all response variables are equal to PAUSED. Hence C_u is dialog-acknowledged and (u, Q) sets $u.s :=$ PAUSED in Rule Q3 within one round. When called, $action_Q(v)$ does not modify $u.p$ since $u.q =$ PAUSED

Case d) $u = v_f \wedge (u.s, u.q) \in \{(\text{PAUSED}, \text{PAUSED}), (\text{PAUSED}, \text{REPAIRED})\}$. All response variables have the value PAUSED and $u.s =$ PAUSED holds in c_0. Furthermore, C_u is dialog-consistent since $(v.s, v.q)$ is a valid query. Hence, the claim holds.

Case e) $u = v_f$, other values of $(u.s, u.q)$. C_u is not dialog-consistent. All responding instances are disabled. Rule Q1 of (u, Q) resets $u.s$ and $u.q$ to PAUSED within the first round. Note that u is now dialog-paused, as none of the responding variables are corrupted. The move of (u, Q) does not modify $u.p$. □

For the remainder of this section, let c_r denote the first configuration in which cell C_{v_f} is repaired. Recall that the initial configuration c_0 is 1-faulty. The following proofs discuss properties of the prefix $e_{pre} = \langle c_0, c_1, \ldots, c_r \rangle$, which is the part of the execution before cell C_{v_f} has become repaired, and the remaining suffix $e_{post} = \langle c_r, c_{r+1}, c_{r+2}, \ldots \rangle$.

Lemma 5.29. *None of the moves within e_{pre} change the backups within cell C_{v_f}. The move that yields c_r (for $r > 0$) is the first modification of $v_f.p$ in e_{pre}. c_r is reached within the first* 4 *rounds of e.*

Proof. By Lemma 5.28, C_{v_f} becomes dialog-consistent and unacknowledged with $v_f.s = \text{PAUSED}$ without modifying $v_f.p$ or any of the backups of $v_f.p$ within 1 round. If $v_f.p$ is not corrupted in c_0, then C_{v_f} is backup-consistent and thus repaired.

If $v_f.p$ is corrupted in c_0, then $\mathit{startCond}_Q(v_f)$ is true and C_{v_f} starts a new cycle. The responding instances provide the acknowledgments for the transition to REPAIRED. These moves do not modify any backups or $v_f.p$. The subsequent move by (v_f, Q) that completes the transition to REPAIRED is the move that yields c_r and is the first to change the primary variable of v_f. By Corollary 5.17 this takes at most 3 more rounds. □

Lemma 5.30. *Within e_{pre} no cell C_u, $u \in N(v_f)$ changes its primary variable or any of its backups.*

Proof. By Lemma 5.28, C_u becomes dialog-consistent and unacknowledged with $u.s = \text{PAUSED}$ without changing its primary variable or the backups. Assume that C_u starts a cycle. If $v_f.b_u$ has not been corrupted, then C_u is still backup-consistent. Hence, the condition in line 3 of $\mathit{action}_Q(u)$ true. Otherwise, if $v_f.b_u$ is corrupted in c_0, the following holds for the transition from PAUSED to REPAIRED:

Case a) $\deg(u) > 1$. The check in line 4 of $\mathit{action}_Q(u)$ prevents a change of $u.p$. The other conditions in lines 5 and 6 are false since $\deg(u) > 1$.

Case b) $\deg(u) = 1$. Since it is assumed that the topology does not consist of a single edge only, there exist at least one neighbor $w \in N(v_f)$. By Lemma 5.29, the value of $v_f.p$ and $w.b_{v_f}$ are not changed before c_r. If $v_f.p$ is corrupted in c_0, then the condition in line 3 of $\mathit{action}_{R_u}(v_f)$ is false since $w.b_{v_f} \neq v_f.p$. If $v_f.p$ is not corrupted, then the condition is false since $G_{\mathcal{A}}(v_f)$ is not satisfied. Hence, $v_f.d_u$ is set to KEEP in line 4 of $\mathit{action}_{R_u}(v_f)$. Hence, $\mathit{action}_Q(u)$ does not modify $u.p$.

If C_u attempts a transition to EXECUTED before c_r, then C_u becomes blocked. Hence, C_u cannot modify $u.p$ during that transition since C_{v_f} is not repaired. Any acknowledgment of the query for a transition to EXECUTED does not modify any backup of $u.p$. Hence, the claim holds. □

Lemma 5.31. *A cell C_u, $u \notin N[v_f]$ is dialog-paused, backup-consistent and disabled in all configurations of e_{pre}.*

Proof. Any cell C_u, $u \notin N[v_f]$ is dialog-paused and backup-consistent in c_0 and u is disabled with respect to $\mathcal{A}$ in c_0. C_u starts a cycle only if $G_{\mathcal{A}}(u)$ is satisfied. However, by Lemma 5.30, none of the cells C_w with $w \in N(v_f)$

modify their primary variable in e_{pre}. Hence, all nodes $u \notin N[v_f]$ remain disabled with respect to $\mathcal{A}$ and C_u remains disabled. □

Lemma 5.32. *Configuration c_r is reached within at most 4 rounds and $\mathcal{L}^*_{\mathcal{A}}$ is satisfied in c_r, i.e., in c_r all nodes are disabled with respect to $\mathcal{A}$.*

Proof. By Lemma 5.28, C_{v_f} becomes dialog-consistent and unacknowledged at position $v_f.s = \text{PAUSED}$ without changing $v_f.p$ or any of the backups of $v_f.p$ within at most one round. If $v_f.p$ is not corrupted in c_0, then cell C_{v_f} is repaired and c_r is the first configuration in which C_{v_f} is dialog-consistent. Otherwise, if $v_f.p$ is corrupted in c_0, C_{v_f} is not backup-consistent after it has become dialog-consistent and thus starts a new cycle. The move of (v, Q) that sets $v_f.s = \text{REPAIRED}$ and thus finishes the transition to REPAIRED yields c_r. By Corollary 5.17 this takes at least 3 more rounds. The following two cases show that $\mathcal{L}^*_{\mathcal{A}}$ holds in c_r.

Case a) $\deg(v_f) > 1$. When called by (v, Q), line 4 of $action_Q(v)$ restores $v_f.p$ to the value in $c_{\mathcal{L}}$ using the backups. By Lemmas 5.30 and 5.31, no node $u \neq v_f$ has changed its primary variable in e_{pre}. Hence, $\mathcal{L}^*_{\mathcal{A}}$ is satisfied in c_r.

Case b) $\deg(v_f) = 1$. Let u denote the neighbor of v_f. Since it is assumed that the topology does not consist of a single edge only, $\deg(u) > 1$. Thus $u.d_{v_f}$ is not set to KEEP in line 2 in $action_{R_{v_f}}(u)$. By Lemma 5.30, $u.p$ and the backups thereof have not been modified. Hence, C_u is nearly backup-consistent, with the exception of $v_f.b_u$ when $action_{R_{v_f}}(u)$ is invoked. By Lemma 5.31, no neighbor of u has modified its primary variable. If both nodes u and v_f are disabled with respect to $\mathcal{A}$, then $u.d_{v_f}$ is set to KEEP in line 4 of $action_{R_{v_f}}(u)$. Hence, when called by (v, Q), $action_Q(v_f)$ does not modify $v_f.p$. Otherwise, $u.d_{v_f}$ is set to UPDATE in line 3 and $v_f.p$ is overwritten with $u.b_{v_f}$ when $action_Q(v_f)$ is called by (v, Q). It is thereby restored to the value of $v_f.p$ in $c_{\mathcal{L}}$. Subsequently, nodes v_f and u are disabled with respect to $\mathcal{A}$. By Lemma 5.31, all other nodes $w \notin \{v_f, u\}$ are disabled with respect to $\mathcal{A}$ as well. Hence, $\mathcal{L}^*_{\mathcal{A}}$ is satisfied in c_r. □

Lemma 5.33. *Let c_i denote a configuration of e_{post} and let all c_j with $r \leq j \leq i$ satisfy $\mathcal{L}^*_{\mathcal{A}}$. A move of a cell C_u with $u \in N[v_f]$ at c_i does not modify $u.p$. A cell C_u, $u \notin N[v_f]$ is dialog-paused, backup-consistent and disabled in c_i.*

Proof. If $u = v_f$, then C_u is repaired in c_r by definition of c_r. Hence, C_u is also repaired in c_i by Lemma 5.21 and the claim follows from Lemma 5.22 as node u is disabled with respect to $\mathcal{A}$.

A cell C_u, $u \notin N[v_f]$ is dialog-paused and backup-consistent in c_r by Lemma 5.31. $startCond_Q(u)$ is true only if u is enabled with respect to $\mathcal{A}$. Hence, C_u does not start a new cycle as $\mathcal{L}^*_{\mathcal{A}}$ is satisfied for all c_j, $r \leq j \leq i$, and thus C_u is dialog-paused and backup-consistent and disabled in c_i.

It remains to discuss $u \in N(v_f)$. It is shown that $u.p$ is not modified by $action_Q(u)$ during the transition from PAUSED to REPAIRED, as $v.q =$ REPAIRED is the only case in which $u.p$ is modified by means other than executing moves of $\mathcal{A}$. If C_u is repaired in c_i, then the claim holds by Lemma 5.22.

For the case that C_u becomes repaired subsequent to c_i, the proof is analogous to the proof of Lemma 5.30 with one exception: If $\deg(u) = 1$, then it needs to be shown that $v_f.d_u$ is set to KEEP in the case that its value is determined by $action_{R_u}(v_f)$ after c_r. This is clearly true, as the condition in line 3 is false since nodes u and v_f are disabled with respect to $\mathcal{A}$ in all c_j, $r \leq j \leq i$. □

Lemma 5.34. *$\mathcal{L}^*_{\mathcal{A}}$ is satisfied for all configurations in e_{post}. A cell C_u, $u \notin N[v_f]$ is dialog-paused, backup-consistent, and disabled in all configurations of e_{post}.*

Proof. It is shown via induction that $\mathcal{L}^*_{\mathcal{A}}$ is satisified for all configurations of e_{post}. The claim follows by Lemma 5.33.

Let c_i, $i \geq r$ denote a configuration of e_{post}. Assume that $\mathcal{L}^*_{\mathcal{A}}$ is satisfied for all configurations $c_r, c_{r+1}, \ldots, c_i$. By Lemma 5.32, this holds for $i = r$. By Lemma 5.33, no primary variables are changed by a move in c_i. Hence, all primary variables in c_{i+1} have the same values as in c_i and thus $\mathcal{L}^*_{\mathcal{A}}$ is satisfied for all configurations $c_r, c_{r+1}, \ldots, c_{i+1}$. □

Theorem 5.35 (Closure)**.** *Let c_i be the first configuration of e such that $\mathcal{L}_{\mathcal{A}}(c)$ is true. Then $\mathcal{L}_{\mathcal{A}}$ holds for all subsequent configurations.*

Proof. By Lemmas 5.29 and 5.30, either $i = 0$ or $i = r$. If $i = 0$, then $\mathcal{L}_{\mathcal{A}}$ is true for all configurations c_j with $0 \leq j < r$. By Lemmas 5.32 and 5.34, $\mathcal{L}_{\mathcal{A}}$ is true for all configuration c_j with $j \geq r$. □

Theorem 5.36 (Contamination Number)**.** *The contamination number of $\mathcal{A}_{FL}$ is at most 1.*

Proof. The contamination number follows from Lemmas 5.30 to 5.32. □

Theorem 5.37 (Containment Time)**.** *The containment time of $\mathcal{A}_{FL}$ is at most 4 rounds.*

Proof. By Lemma 5.32, $\mathcal{L}_{\mathcal{A}}$ is satisfied after at most 4 rounds. □

Theorem 5.38 (Fault-Gap)**.** *The fault-gap of $\mathcal{A}_{FL}$ is at most* 8 *rounds.*

Proof. By Lemmas 5.31 and 5.34, all cells C_u, $u \notin N[v_f]$ are always disabled. We proceed to show that any cell C_u, $u \in N[v_f]$ starts and finishes at most one cycle which takes at most 8 rounds.

By Lemma 5.28, all cells are dialog-consistent at position PAUSED after at most 1 rounds. All cells have finished the transition to position REPAIRED within 3 more rounds by Corollary 5.17. Thus, all cells are repaired and no cell can become blocked. Note that any cell C_u, $u \in N(v_f)$ that finishes the transition to REPAIRED before c_r is blocked and thus cannot finish the transition to EXECUTED before c_r. Hence, no cell C_u, $u \in N(v_f)$ completes its cycle before c_r. By Lemma 5.29, c_r is reached after at most 4 rounds and all cells finish their current cycle within 4 subsequent rounds by Corollary 5.16. As $\mathcal{L}^*_{\mathcal{A}}$ is satisfied by Lemma 5.34 and all cells C_u, $u \in N[v_f]$ are repaired and thus backup-consistent after completing their first cycle, $startCond_Q(u)$ is false for all of them. Hence, $\mathcal{A}_{FL}$ has terminated. □

Theorem 5.39 (Fault-Impact)**.** *The fault-impact of $\mathcal{A}_{FL}$ is $\mathcal{O}(\Delta^2)$ and a radius of at most 2.*

Proof. By Lemmas 5.31 and 5.34, a cell C_u with $u \notin N[v_f]$ is disabled at all times. □

Proof of Theorem 5.11. By Theorem 5.35, $\mathcal{L}_{\mathcal{A}}$ is stable for any execution that starts in a 1-faulty configuration. The containment-time, contamination number, fault-gap, and fault-impact follow from Theorems 5.36 to 5.39. □

5.8.5.3 Serialization

The following ranking $r : \mathcal{I}_{\mathcal{A}_{FL}} \to \mathbb{N}$ is used to obtain partial serializations for steps of $\mathcal{A}_{FL}$ under the distributed scheduler. For notational convenience, the Boolean predicate $e_R(x)$ is used in the definition of $r(v, p)$. It is satisfied if and only if (v, p) is a responding instance (i.e., $p \neq Q$) and the

x-th rule of (v, p) is enabled. If $e_R(x)$ is true, then u refers to the center instance (u, Q) that the responding (v, p) interacts with.

$$r(v,p) := \begin{cases} 1 & \text{if } e_R(2) \wedge u.q = \textsc{Repaired} \\ 2 & \text{if } e_R(1) \vee (e_R(2) \wedge u.q \neq \textsc{Repaired}) \\ 3 & \text{if } p = Q \text{ and } (v, Q) \text{ is enabled} \\ \bot & \text{otherwise} \end{cases}$$

For any step S of $\mathcal{A}_{FL}$ under the distributed scheduler, the ranking r is used to construct partial serializations of the form

$$\langle m_1, m_2, \ldots, m_k, S_{k+1} \rangle$$

where the sequence $m_1, m_2, \ldots, m_k$ consists of all instances within S that have ranks 1 and 2 in c, sorted in ascending order by their rank. The set $S_{k+1} \subseteq S$ is the set of all instances that have rank 3 in c.

Instances of rank 1 set the decision variables. Their value is determined by the evaluation of $G_{\mathcal{A}}$ and by looking at the value of backups. Hence it is important that the primary variables and the backups have not been changed yet. This is only done by instances of rank 2 and 3. Lemma 5.43 proves the correctness of the partial serialization. In the following proofs, the set of variables that differ between c and $(c : m)$ with $m \in \mathcal{I}_{\mathcal{A}_{FL}}$ are called the *changeset* of m at c.

Lemma 5.40. *If cell C_v is not dialog-consistent, then (v, Q) is invariant under any (u, R_v) with $u \in N(v)$.*

Proof. Let c be a configuration in which both $m_2 = (v, Q)$ and $m_1 = (u, R_v)$ are enabled. Let c' denote the configuration $(c : m_1)$.

Rules Q2 and Q3 of m_2 require $\mathit{dialogConsistent}(v)$ and are thus disabled in c. Only Rule Q1 of m_2 is enabled in c and hence $c \vdash (v.s, v.q) \neq (\textsc{Paused}, \textsc{Paused})$. In conclusion, Rule R1 of m_1 is disabled and Rule R2 must be enabled in c. Thus $c \vdash (\mathit{validQuery}(v) \wedge u.r_v = v.s)$. From this and $c \vdash \neg \mathit{dialogConsistent}(v)$ it follows that a neighbor $w \in N(v)$ with $c \vdash w.r_v \notin \{v.s, v.q\}$ exists. Hence $w \neq u$ and $c'|_w = c|_w$ which implies $c' \vdash \neg \mathit{dialogConsistent}(v)$ and $c' \vdash r(m_2) = c \vdash r(m_2)$. Rule Q1 of m_2 sets both $v.s$ and $v.q$ to PAUSED. Hence $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$. □

Lemma 5.41. *If cell C_v is dialog-consistent, then (v, Q) is invariant under any (u, R_v) with $u \in N(v)$.*

Proof. It is shown that no configuration exists in which both (v, Q) and (u, R_v) are enabled. If *dialogConsistent*(v), then Rule Q1 of (v, Q) and Rule R1 of (u, R_v) are disabled. If Rule R2 of (u, R_v) is enabled, then *validQuery*(v) and $u.r_v = v.s$ holds. Hence, neither *dialogPaused*(v) nor *dialogAcknowledged*(v) are true and thus Rules Q2 and Q3 of (v, Q) are disabled. □

Lemma 5.42. *Let C_v and C_w denote two cells with $w \neq v$. Then (v, Q) is invariant under any (u, R_w), $u \in N(w)$.*

Proof. By Observation 4.25, only the case $u \in N[v]$ needs to be considered. At any configuration, the changeset of (u, R_w) is a subset of $\{u.r_w, u.d_w, u.b_w\}$. Rank, guards and statements of (v, Q) only depend on variables from cell C_v and primary variables of neighboring cells. These are not contained in the changeset of (u, R_w). □

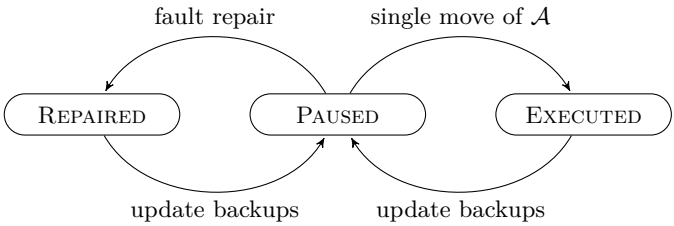
Lemma 5.43. *The ranking r is a weak invariancy ranking. For instances of rank less than 3, r is a proper invariancy-ranking.*

Proof. Let m_2 and m_1 with $m_2 \neq m_1$ denote two instances of nodes v_2 and v_1 and cells u_2 and u_1 respectively. For the given ranks r_2, r_1, let c denote a configuration which satisfies $r_2 = c \vdash r(m_2)$ and $r_1 = c \vdash r(m_1)$ and c' denotes the configuration $(c : m_1)$. By Observation 4.25, only the case that m_2 and m_1 are neighboring instances is considered.

First, it is shown that r is a proper invariancy ranking for instances of rank less than 3:

Case a) $r_2 = 1 \wedge r_1 = 1$: The assumption yields that Rule R2 of both m_2 and m_1 are enabled in c and that $c \vdash u_1.q = u_2.q = \textsc{Repaired}$. Since $c \vdash u_1.q = \textsc{Repaired}$, the changeset of m_1 at c is $\{v_1.r_{u_1}, v_1.d_{u_1}\}$. Hence, $c' \vdash u_2.q = \textsc{Repaired}$ holds and thus the guard of Rule R2 of m_2 only depends on the variables $v_2.r_{u_2}$, $u_2.s$, and $u_2.q$ which are not in the changeset of m_1. Thus Rule R2 of m_2 remains enabled in c' and thus $c' \vdash r(m_2) = c \vdash r(m_2)$. Since $c' \vdash u_2.q = \textsc{Repaired}$, the output of $(c' : m_2)|_{m_2}$ only depends on the variable $u_2.q$, primary variables, and backups which are not part of the changeset of m_1.

Case b) $r_2 = 2 \wedge r_1 \in \{1, 2\}$: At worst, the changeset of m_1 is $\{v_1.r_{u_1}, v_1.b_{u_1}, v_1.d_{u_1}\}$. The predicate *repaired*(v_2) is the only part of the guards of m_2 which might reference these variables, namely in the case $v_2 = u_1$. By Lemma 5.21 $c \vdash$ *repaired*$(v) \Rightarrow c' \vdash$ *repaired*(v). Hence the rule of m_2 that is enabled in c is still enabled in c'. It follows that $c' \vdash r(m_2) = c \vdash r(m_2)$. Since m_2 is of rank 2, it sets $v_2.r_{u_2}$ to $u_2.q$ and possibly updates $v_2.b_{u_2}$



$$\begin{aligned} \mathit{validQuery}(s,q) \equiv\ & (s,q) \in (\{\textsc{Paused}\} \times \{\textsc{Repaired}, \textsc{Executed}\}) \lor \\ & (s,q) \in (\{\textsc{Repaired}, \textsc{Executed}\} \times \{\textsc{Paused}\}) \\ \mathit{dialogConsistent}(v) \equiv\ & \mathit{dialogPaused}(v) \lor (\mathit{validQuery}(v.s, v.q) \land \\ & \forall u \in N(v) : u.r_v = v.s \lor \mathit{validQuery}(v.s, u.r_v)) \end{aligned}$$

Figure 5.16: Optimized state-machine for cells

with $u_2.p$. Both $u_2.q$ and $u_2.p$ are not among the changeset of m_1 and thus $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$.

The above shows that r is also a weak invariancy-ranking for instances of rank less than 3. It remains to show that this also holds if instances of rank 3 are involved.

Case c) $r_2 = 3 \land r_1 \in \{1, 2\}$: If $u_1 \neq u_2$, then Lemma 5.42 applies. Otherwise either Lemma 5.41 or Lemma 5.40 applies. □

5.8.6 Optimizations and Modifications

It is possible to optimize the results in Section 5.8.5. In this section we provide short sketches of these optimizations.

By using an alternative state-machine, it is possible to reduce the fault-gap of $\mathcal{A}_{FL}$ to 4 rounds and the slowdown-factor to 4. The alternative state-machine is shown in Figure 5.16. It consists of two cycles with two transitions each. At position Paused, a cell should then decide whether to execute a repair cycle (left) or convergence cycle (right) depending on whether it is backup-consistent or not. However, starting in a 1-faulty configuration, a cell C_v may attempt starting a convergence cycle if $(v.s, v.q) = (\textsc{Paused}, \textsc{Executed})$ in the initial configuration. It is therefore necessary to allow a cell to correct this decision, even though some of the responding instances may have already acknowledged the query for a transition to

EXECUTED. A compatible definition of dialog-consistency that allows this is given in Figure 5.16. Rule R2 must be changed in such a way that it provides a new acknowledgment after the center instance changes the query. In particular, procedure $action_{R_v}$ must be invoked, so that the decision-variables can be set.

The transformation only relies on weakly unique identifiers, mainly so that the neighbor to which variables of a responding instance belong can be identified. The only exception is the procedure *pairRepair*, shown in Figure 5.15, that handles the single-edge case described in Section 5.8.4.3. Under the central scheduler, a much simpler version that does not rely on locally unique identifiers can be used. Such an implementation of *pairRepair* would simply overwrite $v.p$ with $u.b_v$ whenever $pairEnabled_{\mathcal{A}}(v : v.p, u : u.p)$ and not $pairEnabled_{\mathcal{A}}(v : u.b_v, u : u.p)$.

To transform algorithms that stabilize with only weakly unique identifiers under the distributed scheduler, a different solution for handling the single-edge case must be found. For such algorithms, a symmetrical legitimate configuration, where the local state of both nodes is identical, must exist in the case where the topology consists of a single edge. It is obtained by starting $\mathcal{A}$ under the synchronous scheduler in a symmetrical initial configuration. This can be used by $pairRepair(v, u)$ as follows: If a symmetrical 1-faulty configuration is detected, the value of $v.p$ is set to a fixed value x that satisfies $\neg pairEnabled_{\mathcal{A}}(v : x, u : x)$. This ensures that both nodes set their primary variable correctly when *pairRepair* is invoked by both nodes simultaneously. If *pairRepair* is not invoked simultaneously by both nodes, the second call to *pairRepair* can detect that a reset to x has been initiated.

The transformation can handle multiple state corruptions without any modifications if the distance between any pair of faulty nodes is at least 3. Then, at most one instance per cell is affected by the state corruptions, and the fault repair implemented in $action_Q$ and $action_{R_v}$ succeeds. It is also possible to handle multiple corruptions within the same cell. For that to be possible, $action_Q$ must be changed in such a way that it uses the value that is in a majority among $v.p$ and its backups for repair. Furthermore, the logic in $action_{R_v}(u)$ that handles the case where v is a leaf node must be adjusted so that it anticipates the value that $u.p$ will be overwritten with by $action_Q(u)$. Note that for a leaf node v the case where both $v.p$ and $u.b_v$ are corrupted is not handled. Hence, if an area with many nodes with low degree is hit by several faults, there is a high risk that the faults cannot be contained.

5.9 Discussion

This chapter has presented several techniques for designing fault-containing self-stabilizing distributed algorithms. The first technique, a transformation based on global synchronization, has scalability problems. As the fault-gap depends on the number of nodes and diameter of the system's topology, the time between containable faults grows as systems become larger. Moreover, the larger the number of nodes, the higher the probability of faults.

Several attempts to lower the fault-gap, and in particular to make it independent of the number of nodes, have been made: the design of problem-specific solutions, probabilistic fault-containment, and priority scheduling. However, an impossibility result by Ghosh et al. shows that a transformation cannot provide both a fault-gap of $\mathcal{O}(1)$ rounds and the stabilization time of $T + \mathcal{O}(1)$ rounds, where T is the stabilization time of $\mathcal{A}$ in rounds [GGHP96]. Section 5.8 presented a new transformation that utilizes local instead of global synchronization. Its output algorithm $\mathcal{A}_{FL}$ combines a fault-gap of $\mathcal{O}(1)$ and a fault-impact of radius 2 with a stabilization time of $\mathcal{O}(T)$. The transformation scales well, as the fault-gap does not depend on the number of nodes or the degree of the nodes. It is therefore suitable for large networks. Also the small fault-impact radius indicates that most parts of the system continue to operate and are completely unaware of the fault. These results are considerable improvements over the previously known transformation which is based on global synchronization. Also, $\mathcal{A}_{FL}$ does not require any a priori knowledge about n or Δ.

The fault-containment provided by $\mathcal{A}_{FL}$ is comparable to that provided by the problem-specific solutions and the priority scheduling based approach. However, priority scheduling seems to cause a large slow-down and it is an open problem whether it can be the basis of a transformation. The stabilization time of $\mathcal{A}_{FL}$ differs asymptotically from that of the input algorithm $\mathcal{A}$ by a constant slow-down factor of 6 only. We assume that the key aspect of the efficiency of $\mathcal{A}_{FL}$ is local synchronization based on stable properties (see Lemma 5.21). The priority scheduler, however, must ensure that the priority of a neighbor of v never increases while v makes a move. Hence, the entire 2-hop neighborhood of v is prohibited to make a move. Thus a non-constant slow-down may not be avoidable.

Unlike the transformation for 1-strong stabilization discussed in Section 5.3.2, $\mathcal{A}_{FL}$ is correct under the distributed scheduler. This allows the use of $\mathcal{A}_{FL}$ in asynchronous systems without the need for synchronizers. However, it should be mentioned that even if $\mathcal{A}_{FL}$ is executed under the synchronous scheduler, then $\mathcal{A}$ must be assumed to be silent and self-

stabilizing under the distributed scheduler. This stems from the fact that the moves of one round of $\mathcal{A}$ are spread out over 6 rounds of $\mathcal{A}_{FL}$.

Compared to $\mathcal{A}$, the space required by $\mathcal{A}_{FL}$ increases by a factor of $\Delta + 1$. The required space is comparable to the previously known transformation. However, it is still a disadvantage compared to specially designed problem-specific solutions. It can be shown that only 2 backups per node are sufficient to repair the state corruption of a single node. Lowering the number of backups from Δ to 2 yields a new problem: Which neighbors should a node choose to store its backups? Chapter 6 discusses a possible solution to this problem which computes a placement for the backups such that the number of backups stored by each node has minimal variance.

All solutions to fault-containment discussed in this chapter assume that the topology remains unchanged. Fault-containment, i.e., repair of a state corruption in constant time, is discussed in Chapter 7.

6 Local k-Placement with Local Minimum Variance

Large-scale distributed systems such as cloud computing networks, peer-to-peer file sharing systems, or sensor networks, often require replication of resources. Replicas are placed on other nodes of the network. There are different reasons to create replicas: reduction of retrieval time (e.g., caching systems), better utilization of computing devices (e.g., load balancing), or the increase of fault tolerance and availability (e.g., backup systems). The placement strategy for the replicated resources significantly impacts performance in all these cases.

This chapter considers distributed systems where each node wants to place replicas of a resource on k different neighbors. The nature of the resource is of no importance for our discussion. However, it is assumed that all replicas induce the same load. As an example consider the case where each node wants to replicate its own state on k neighbors. The motivation for this stems from the transformation presented in Chapter 5. Adding fault-containment to any silent self-stabilizing protocol can be achieved by creating backups of a node's local state on $k = 2$ neighbors.

We pursue the goal of achieving a preferably homogeneous placement of the replicas. There are different measures of homogeneity of a placement, e.g., the Gini coefficient, the standard deviation, or the discrepancy, which is the difference between the maximum and the minimum load among all nodes. In many cases the network topology will not permit having a completely homogeneous placement, i.e., the same load for every node resp. a standard deviation of 0. Therefore, finding a distributed algorithm for this placement problem is a real challenge. The problem is formally defined in Section 6.1. Related problems are discussed in Section 6.2.

The main contribution of this chapter is a silent self-stabilizing distributed algorithm, designed for the unfair distributed scheduler, that computes the placement of k replicated resources on direct neighbors of each node, such that the standard deviation of the number of replicas per node assumes a local minimum. The algorithm is described in Section 6.4. It terminates after $O(n\Delta^2)$ moves or $O(n\Delta)$ rounds respectively under the distributed

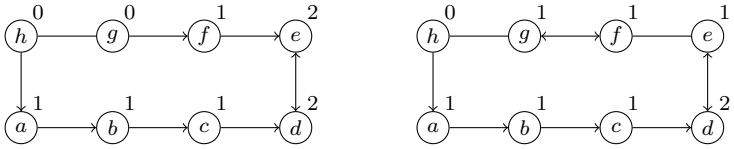


Figure 6.1: Two local 1-placements of the same graph

scheduler. This chapter concludes with a sketch of how this algorithm can be integrated with the transformation given in Section 5.8.

6.1 The Local *k*-Placement Problem

For any given positive constant $k \in \mathbb{N}$, a local k-placement selects k neighbors per node v that store one of k replicas of a resource of node v. For nodes $v \in V$ with less than k neighbors, we restrict the number of replicas to $\deg(v)$ and all neighbors are selected. For notational convenience we define $k(v) = \min(\deg(v), k)$ for all $v \in V$. Consider the following formal definition:

Definition 6.1. A *local k-placement* is a function β that assigns to each $v \in V$ a set $\beta(v) \subseteq N(v)$ with $|\beta(v)| = k(v)$. The *load* of v with respect to β is denoted by $L_\beta(v) = \#\, w \in N(v) : v \in \beta(w)$.

Informally, the load of node v is the number of times that node v is selected to store a replica for one of its neighbors. Figure 6.1 depicts two different local 1-placements of the same graph. Arrows indicate the placement of the replica of each node. The label next to each node indicates its load.

The goal pursued in this chapter is to find a homogeneous distribution of the load among the nodes. There are several expressions to measure the homogeneity of a local k-placement β. We have chosen the standard deviation, or equivalently the variance, which is defined as follows:

$$var(\beta) = \frac{1}{n} \sum_{v \in V} (L_\beta(v) - \mu_\beta)^2 \qquad \text{with } \mu_\beta = \frac{1}{n} \sum_{v \in V} L_\beta(v).$$

The lower the value of $var(\beta)$ the more homogeneous is β. The local 1-placement on the left of Figure 6.1 has a variance of $1/2$ while that on the right has variance $1/4$.

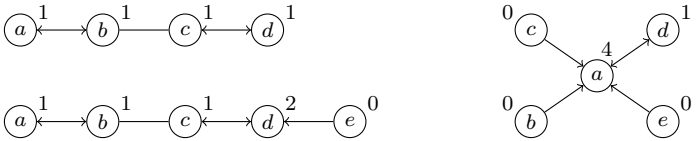


Figure 6.2: Local 1-placements with (global) minimum variance: Line graphs of odd and even length (left) and a star graph (right)

Computing a local k-placement with minimum variance distributedly seems to be a difficult task. This is discussed in more detail in Section 6.3. Therefore, we content ourselves with computing local minima. A local k-placement has *local minimum variance* if the reassignment of a single replica from one neighbor to another does not lower the variance. The local 1-placement on the left of Figure 6.1 does not have local minimum variance. If node f shifts its replica from node e to g the variance is decreased from $\frac{1}{2}$ to $\frac{1}{4}$. The result is the local 1-placement shown on the right. This placement has local minimum variance. Note that an optimal local k-placement of this graph has variance 0. Obviously, a variance of 0 is the (global) minimum in this case. Not all graphs have a local k-placement with variance 0. Examples of such graphs are line graphs with an odd number of nodes and star graphs, see Figure 6.2.

6.2 Related Problems

The local k-placement restricts the placement of replicas of a node to the 1-hop neighborhood of a node. To the best of our knowledge, a similarly restricted placement of resources does not seem to have been discussed yet in the literature. Hence, this section gives an overview of work on problems related to balancing load in terms of storage, CPU time, and network I/O and on problems related to unrestricted placement of resources.

6.2.1 Balancing Load

Since the early days of distributed computing load distribution has received a lot of attention and many schemes have been devised. In general the term refers to algorithms improving the overall system performance by transferring some form of load (e.g., a task or a resource) from heavily to lightly loaded nodes. This section only reviews work with strong links to the local

k-placement problem. Gärtner et al. present a method for providing load balancing for replicated servers on a per access basis [GP99]. In this case the replicated resources are servers, i.e., processes and data. But instead of moving resources between nodes, accesses to the resources are redirected to achieve a homogeneous access distribution. The self-stabilization property of the algorithm is achieved by a composition of two distributed algorithms.

Two self-stabilizing algorithms for migrating the job load around the network are presented by Flatebo et al. [FDB94]. In the first algorithm each node compares its own load with the load of its neighbors. When a neighbor momentarily has a lower load, the node migrates a job to this neighbor. The second algorithm aims at migrating jobs globally to lightly loaded nodes. The run-time complexity of the algorithms is not considered.

Sauerwald and Sun consider the problem of balancing the number of tokens per node [SS12]. Each token represents a load. Starting with an arbitrary token distribution, in each round nodes exchange tokens with their neighbors. The goal is to achieve a distribution where all nodes have nearly the same number of tokens. The authors present a detailed analysis of their randomized algorithms. Lenzen and Wattenhofer recently determined tight bounds for a related task: the so-called balls-into-bins problem [LW11].

Load balancing has received a lot of attention in the field of peer-to-peer networks. The problem arises in two flavors: balancing the distribution of the key address space to nodes (in DHT-based systems) and balancing the distribution of items among the nodes. Karger and Ruhl address both problems in the context of a Chord DHT [KR06]. Serbu et al. present a solution that aims to equilibrate the request load and the routing load of the nodes in DHT-based peer-to-peer systems where requests follow a Zipf-like distribution [SBKF07]. Instead of changing the placement of objects the routing tables are reorganized.

6.2.2 Optimized Placement

Ko et al. present a distributed algorithm that places replicated resources in a network among all nodes such that the furthest distance to a particular replica of a resource is minimized [KR05]. The algorithm guarantees an approximation of ratio 3. No analysis of the time complexity is given. A similar problem is tackled by Kangasharju et al. They consider techniques for optimally replicating objects in content distribution networks [KRR02]. The goal is to replicate objects on servers with finite storage so that when clients fetch objects from the nearest server holding the requested object,

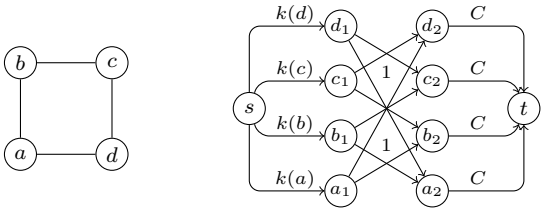


Figure 6.3: A graph G (left) and the corresponding graph $G_{k,C}$ (right)

the average number of nodes traversed is minimized. They show that this optimization problem is NP-complete and present heuristics.

6.3 Reductions to Flow Problems

In this section, we discuss sequential algorithms for finding a local k-placement with (global) minimum variance for a given graph $G = (V, E)$. In addition, the *capacity-bounded* local k-placement problem is discussed: Given a capacity of $C \in \mathbb{N}$ replicas per node, does a local k-placement β exist such that $L_\beta(v) \leq C$ for all nodes $v \in V$? Both algorithms are based on a reduction to flow problems.

Both reductions to flow problems involve the construction of a directed bipartite graph $G_{k,C}$. For a given graph $G = (V, E)$, the graph $G_{k,C}$ consists of two copies of V and two additional nodes: a source node s and a target node t. Let v_1 and v_2 denote the first and second copy of a node $v \in V$. In $G_{k,C}$ there are edges with capacity $k(v)$ from s to v_1 and an edge with capacity C from v_2 to t for each $v \in V$. For every edge (u, v) of G, $G_{k,C}$ contains an edge with capacity 1 from u_1 to v_2 and v_1 to u_2. The construction is illustrated in Figure 6.3. The labels to the edges denote their capacity.

Observation 6.2. There exists a local k-placement with $L_\beta(v) \leq C$ for all $v \in V$ if and only if the maximum flow from s to t in $G_{k,C}$ has the value $\sum_{v \in V} k(v)$.

Note that $u \in \beta(v)$ if and only if the flow through the edge (v_1, u_2) of $G_{k,C}$ is 1. Hence, there is a polynomial time sequential algorithm for solving the capacity-bounded local k-placement problem. Repeated application of the algorithm computes the minimal C for which a local k-placement exists. A self-stabilizing algorithm for computing a maximum flow does ex-

ist (e.g., [GGP97b]), but its application makes it necessary to emulate the constructed bipartite graph with additional nodes s and t on the original graph G. Moreover, the time-complexity of this algorithm is unknown.

The problem of computing a k-local placement β with (global) minimum variance can be reduced to a special type of minimum cost flow problem on a directed bipartite graph similar to $G_{k,C}$ described above. There are two differences. Firstly the capacity C is set to $C = \sum_{v \in V} k(v)$. This capacity is large enough such that the placement of the replicas is not restricted by C. Secondly the edges carry costs. The cost associated with an edge adjacent to t is set to the square of the flow through that edge (i.e., a convex function). The costs associated with all other edges is 0. An integral solution to this minimum cost flow problem corresponds to a local k-placement where $\sum_{v \in V} L_\beta(v)^2$ assumes a minimum. As $var(\beta) = \left(\frac{1}{n} \sum_{v \in V} L_\beta(v)^2\right) - \mu_\beta^2$, where μ_β^2 is constant for any local k-placement β, the solution to the minimum cost flow problem also corresponds to a local k-placement with minimum variance. The described flow problem is a convex cost integer dual network flow problem as described by Ahuja et al. in [AHO03]. They also give a sequential algorithm with polynomial complexity for this problem. To the best of our knowledge no non-trivial distributed algorithm is known for that problem and the distributed emulation of the graph $G_{k,C}$ might prove to be difficult.

6.4 Placement Algorithm

This section describes the implementation of a self-stabilizing distributed algorithm that computes a local k-placement problem β as defined in Section 6.1. The goal of the algorithm is to minimize the standard deviation or equivalently the variance of the load per node. If a node $v \in V$ moves its replica from node $u \in \beta(v)$ to a neighbor $w \in N(v) \setminus \beta(v)$, then we obtain another k-placement β' with $\beta'(v) = (\beta(v) \setminus \{u\}) \cup \{w\}$. It satisfies $L_{\beta'}(u) = L_\beta(u) - 1$, $L_{\beta'}(w) = L_\beta(w) + 1$, and $L_{\beta'}(v) = L_\beta(v)$ for all $v \in V$. Since $\mu_{\beta'} = \mu_\beta$, this yields

$$var(\beta) - var(\beta') = \frac{2}{n} \left(L_\beta(u) - 1 - L_\beta(w)\right) \tag{6.1}$$

Thus, the variance of β' is lower than that of β if and only if $L_\beta(u) > L_\beta(w) + 1$. This yields the basic strategy of the proposed algorithm: Once a node v detects such a pair $u, w \in N(v)$, it moves a replica on u to w.

Each node $v \in V$ maintains a variable $v.\beta$ which is a set of up to k pointers to neighboring nodes. Once the algorithm has stabilized, $v.\beta$ represents the

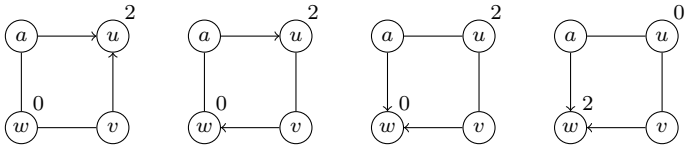


Figure 6.4: Example of oscillation, $k = 1$

value of $\beta(v)$ for the computed local k-placement. Based on variable $v.\beta$, the current load of a node, i.e., the number of replicas placed at the node, is defined as follows:

$$L(v) := \#\, w \in N(v) : v \in w.\beta$$

The basic idea of the algorithm is that each node $v \in V$ periodically checks whether the shifting of one its replicas from one neighbor to another would lower the standard deviation of the load of its neighbors. If such a case is detected, the shift of the replica is initiated. The challenge is to coordinate these movements such that no livelocks occur.

With 1-hop knowledge only, it is impossible for v to determine the value of $L(u)$ for a neighbor $u \in N(v)$. Hence, each node $v \in V$ maintains the so-called *load variable* $v.l$ which is updated with the current value of $L(v)$ whenever possible. The variable $v.l$ can be out-of-date and that can potentially lead to oscillation, even under the central scheduler. An example is illustrated in Figure 6.4. The numbers next to u and w denote the values of $u.l$ and $w.l$. The arrows represent the values of $v.\beta$ and $a.\beta$. The scheduler selects v and a during the first stop steps. Both nodes move their replica from u to w. However, the decision of a to do so is based on outdated values of $u.l$ and $w.l$. Thus, after updating the values of $u.l$ and $w.l$, the situation is symmetric to the initial configuration. Even under a fair central scheduler, this may result in a livelock.

To prevent a livelock, the algorithm uses a synchronization mechanism. Before node $v \in V$ moves a replica from u to w, it locks u with regards to removal and w with regards to addition of replicas. While u and w are locked, it is not allowed for other nodes to move their replica away from u or towards w. Also, during the locking procedure, both $u.l$ and $w.l$ will be updated. Hence, when v has successfully locked u and w it is safe to assume the following:

$$u.l > w.l + 1 \Rightarrow L(u) > L(w) + 1 \tag{6.2}$$

It is, however, still possible for nodes other than v to move their replicas towards u or away from w. Hence, $L(u)$ may grow larger than $u.l$ and $L(w)$ may become less than $w.l$. This has no impact on Equation (6.2). The two locking mechanisms, one for removal and one for addition of replicas, are implemented using two pointer variables each, called *query* and *ack*. In order for node v to lock a neighboring node u with respect to replica additions, v first sets its query $v.q_a := u$. It then waits until u changes its ack to $u.a_a = v$. Once v and u point at each other, u is said to be locked by v. Thus, a node can be locked for addition by at most one neighbor at a time. If no query is made or there is no query to acknowledge, then $v.q_a$ and $u.a_a$ assume the special value $\perp$. In order to lock a neighbor u with respect to replica removal instead of addition, the variables $v.q_r$ and $u.a_r$ are used instead of $v.q_a$ and $u.a_a$.

The proposed algorithm $\mathcal{A}_P$ with $\mathcal{A}_P(v) = \{P_L, P_{A_r}, P_{A_a}, P_Q\}$ for all $v \in V$ uses four protocols. Their implementation is shown in Figure 6.5. Protocol P_L updates the load variable $v.l$. Protocols P_{A_r} and P_{A_a} provide the acknowledgments for the locking mechanisms. Queries for removal (resp. addition) are only acknowledged if the load variable $v.l$ is less or equal (resp. greater or equal) to the actual load. The values of $v.a_r$ and $v.a_a$ are controlled via the sets $A_r(v)$ and $A_a(v)$. They contain all neighbors waiting for an acknowledgment by v. (v, P_{A_r}) updates $v.a_r$ such that $v.a_r \in A_r(v)$ or $v.a_r = \perp$ if $A_r(v) = \emptyset$. (v, P_{A_a}) implements the same for $v.a_a$ and $A_a(v)$. Note that no assumption about the value of $choose(X)$ is made in the case where X contains multiple elements. $choose(X)$ may return any non-deterministically chosen element of X. The definition of the sets $A_r(v)$ and $A_a(v)$ is as follows:

$$A_r(v) = \{w \in N(v) \mid w.q_r = v \wedge v \in w.\beta\}$$
$$A_a(v) = \{w \in N(v) \mid w.q_a = v \wedge v \notin w.\beta\}$$

Protocol P_Q controls the queries and the current placement. Rule P1 of (v, P_Q) is responsible for removing all invalid pointers from $v.\beta$, i.e., pointers that do not refer to a neighbor of v, and for adding valid pointers until $v.\beta$ has reached the desired cardinality $k(v)$. Rule P2 is responsible for setting the query $v.q_r$. The value of the variable $v.q_r$ is controlled via the sets $Q_r(v)$ and $Q'_r(v)$. The former contains those neighbors $w \in N(v)$ with maximal $w.l$ at which v currently places a replica and for which a neighbor $u \in N(v)$ exists which currently does not store a replica such that $u.l < w.l - 1$. $Q'_r(v)$ contains only those $w \in Q_r(v)$ which do not already acknowledge a query by v. This ensures that (w, P_{A_r}) will update $w.l$ before providing

Protocol P_L

Variables: $v.l$: Integer in the range $[0, \Delta]$

do $v.l \neq L(v) \longrightarrow v.l := L(v)$ **od**

Protocol P_{A_r}

Variables: $v.a_r$: Node identifier

do $v.l \leq L(v) \wedge \mathit{updateAck}_r(v) \longrightarrow v.a_r := \mathit{choose}(A_r(v))$ **od**

Protocol P_{A_a}

Variables: $v.a_a$: Node identifier

do $v.l \geq L(v) \wedge \mathit{updateAck}_a(v) \longrightarrow v.a_a := \mathit{choose}(A_a(v))$ **od**

Protocol P_Q

Variables: $v.q_r$, $v.q_a$: Node identifiers
$v.\beta$: set of at most k node identifiers

do

[P1] $\neg \mathit{validPlacement}(v) \longrightarrow$
$v.\beta := v.\beta \cap N(v)$; add $k(v) - |v.\beta|$ pointers to $v.\beta$

Precondition for Rules P2–P5: $\mathit{validPlacement}(v)$

[P2] $\Box\ \mathit{updateQuery}_r(v) \longrightarrow v.q_r := \mathit{choose}(Q'_r(v)); v.q_a := \perp$

[P3] $\Box\ \neg(\mathit{updateQuery}_r(v) \vee \mathit{queryValidAndAcked}_r(v)) \wedge$
$v.q_a \neq \perp \longrightarrow v.q_a := \perp$

[P4] $\Box\ \mathit{queryValidAndAcked}_r(v) \wedge \mathit{updateQuery}_a(v) \longrightarrow$
$v.q_a := \mathit{choose}(Q'_a(v))$

[P5] $\Box\ \mathit{queryValidAndAcked}_r(v) \wedge \mathit{queryValidAndAcked}_a(v) \longrightarrow$
$v.\beta := (v.\beta \setminus \{v.q_r\}) \cup \{v.q_a\}; v.q_r := \perp; v.q_a := \perp$

od

$$\mathit{choose}(X) := \begin{cases} \perp & \text{if } X = \emptyset \\ \text{element of } X & \text{otherwise} \end{cases}$$

$$\mathit{validPlacement}(v) \equiv |v.\beta| = k(v) \wedge v.\beta \subseteq N(v)$$

$$\mathit{updateAck}_x(v) \equiv v.a_x \notin A_x(v) \wedge (A_x(v) \neq \emptyset \vee v.a_x \neq \perp)$$

$$\mathit{updateQuery}_x(v) \equiv v.q_x \notin Q_x(v) \wedge (Q'_x(v) \neq \emptyset \vee v.q_x \neq \perp)$$

$$\mathit{queryValidAndAcked}_x(v) \equiv v.q_x \in Q_x(v) \wedge (v.q_x).a_x = v$$

where $x \in \{a, r\}$

Figure 6.5: Protocols of algorithm $\mathcal{A}_P$

an acknowledgment. The query $v.q_r$ is said to be *valid* if $v.q_r \in Q_r(v)$. If $v.q_r$ is not valid, Rule P2 updates $v.q_r$ such that $v.q_r \in Q'_r(v)$ or $v.q_r = \perp$ if $Q'_r(v) = \emptyset$. Note that Rule P2 also resets $v.q_a$ to $\perp$. As long as the query $v.q_r$ is either invalid or not acknowledged, $v.q_a$ must remain $\perp$ such that locks with respect to addition of replicas are not established before the removal lock. This is essential for deadlock-free operation. Hence, Rule P3 makes sure that $v.q_a$ is reset to $\perp$ to avoid any premature queries, e.g., due to transient faults or initial inconsistencies.

If $v.q_r$ is valid and acknowledged, then Rule P4 controls the value of $v.q_a$ via $Q_a(v)$ and $Q'_a(v)$. The former contains all neighbors $w \in N(v)$ that currently do not hold replicas of v and have minimal $w.l$. Again $Q'_a(v)$ contains only those neighbors $w \in Q_a(v)$ which do not already acknowledge a query of v in order to ensure an update of $w.l$ before the lock is being established.

$$\begin{aligned}
Q_r(v) &= \{w \in N(v) \cap v.\beta \mid w.l = \max\{u.l \mid u \in N(v) \cap v.\beta\} \wedge \\
&\qquad \exists u \in N(v) \setminus v.\beta : u.l < w.l - 1\} \\
Q_a(v) &= \{w \in N(v) \setminus v.\beta \mid w.l = \min\{u.l \mid u \in N(v) \setminus v.\beta\}\} \\
Q'_r(v) &= \{w \in Q_r(v) \mid w.a_r \neq v\} \\
Q'_a(v) &= \{w \in Q_a(v) \mid w.a_a \neq v\}
\end{aligned}$$

If both $v.q_r$ and $v.q_a$ are valid and acknowledged, then Rule P5 is enabled and will move one replica of v from $v.q_r$ to $v.q_a$ by changing $v.\beta$. Note that both queries might be reset via Rules P2 and P4 multiple times as $Q_r(v)$ and $Q_a(v)$ change whenever a neighbor updates its load variables. The following Boolean predicate identifies all legitimate configurations of $\mathcal{A}_P$:

$$\begin{aligned}
\mathcal{L}_{\mathcal{A}_P} \equiv \forall v \in V : &\mathit{validLoad}(v) \wedge \mathit{validPlacement}(v) \\
&\wedge v.q_r = v.a_r = v.q_a = v.a_a = \perp \wedge Q_r(v) = \emptyset
\end{aligned}$$

6.5 Potential Function

In this section, a potential function for $\mathcal{A}_P$ is constructed. First for the central scheduler. Then the potential function is refined for the distributed scheduler using partial serializations. The potential functions will be used to show termination and for determining the move-complexity of $\mathcal{A}_P$ in Section 6.6.

6.5.1 Central Scheduler

In this section we will present the potential function *potvec* that maps $\Sigma_{\mathcal{A}_P}$ to the lexicographically sorted $\mathbb{N}_0^6$. Furthermore, this section will present a technique to transform *potvec* into the potential function *pot* whose co-domain is merely $\mathbb{N}_0$. For the construction of *pot*, the central scheduler is assumed. Hence, the maximum value of *pot* is an upper bound the move-complexity of $\mathcal{A}_P$ under the central scheduler.

$$potvec(c) := \begin{pmatrix} \#\, v \in V : \neg validPlacement(v) \\ \#\, v \in V : malicious(v) \\ sqsum(c) \\ \#\, v \in V : \neg validLoad(v) \\ \sum_{v \in V} progress_Q(v) \\ \sum_{v \in V} progress_A(v) \end{pmatrix}$$

The key component of *potvec* is $sqsum(c) := \sum_{v \in V} L(v)^2$. It decreases by $L(u) - 1 - L(w)$ if a replica is moved from node u to w. It is used as a substitute for the variance (cf. Equation (6.1) and Section 6.3). Since algorithm $\mathcal{A}_P$ is based on the idea to move a replica from u to w whenever $L(u) > 1 + L(w)$, it can be expected that $sqsum(c)$ decreases regularly, namely due to the execution of Rule P5. However, two scenarios exist in which $sqsum(c)$ actually increases.

The first scenario is a node v executing Rule P5, but despite having locked neighbors u and w, either $L(u)$ is less than $u.l$ or $L(w)$ is larger than $w.l$. In this case, $L(u) > L(w) + 1$ might not hold even though $u.l > w.l + 1$ is satisfied. Hence, Equation (6.2) is violated. Still, v moves a replica from u to w based on the outdated values of $u.l$ and $w.l$. Such situations can only arise from inconsistencies in the initial configuration and executions of Rule P1. Nodes which will (depending on the schedule) be able to execute Rule P5 under such circumstances at some future point in the execution are called *malicious* (see the corresponding predicate below). The number of malicious nodes constitutes another component of *potvec*. Depending on the choices of the scheduler, a node may become non-malicious if either $u.l$ or $w.l$ is updated before v executes Rule P5. Also note that a node becomes non-malicious by executing Rule P5 and that the locking mechanism ensures that non-malicious nodes do not become malicious. The only exception is Rule P1 which bypasses the locking mechanism and thereby may increase the number of (potentially) malicious nodes.

An execution of Rule P1 poses the second scenario in which the variance actually increases. To reflect that, the first component of *potvec* denotes the

number of nodes enabled with respect to Rule P1. It decreases whenever a node executes Rule P1 which happens at most once per node during an execution.

$$\begin{aligned} \mathit{malicious}(v) \equiv\ & \mathit{queryValidAndAcked}_r(v) \wedge (L(v.q_r) < (v.q_r).l \vee \\ & (\mathit{queryValidAndAcked}_a(v) \wedge L(v.q_a) > (v.q_a).l)) \end{aligned}$$

$$\mathit{progress}_Q(v) := \begin{cases} 0 & \text{if } \mathit{queryValidAndAcked}_r(v) \wedge v.q_a \in Q_a(v) \\ 1 & \text{if } v.q_r \in Q_r(v) \wedge v.q_a = \bot \\ 2 & \text{if } v.q_r = \bot \wedge v.q_a = \bot \\ 3 & \text{otherwise} \end{cases}$$

$$\mathit{progress}_A(v) := \begin{cases} 0 & \text{if } \neg \mathit{updateAck}_r(v) \wedge \neg \mathit{updateAck}_a(v) \\ 1 & \text{if } \mathit{updateAck}_r(v) \neq \mathit{updateAck}_a(v) \\ 2 & \text{if } \mathit{updateAck}_r(v) \wedge \mathit{updateAck}_a(v) \end{cases}$$

The remaining components of *potvec* are the number of nodes with an out-of-date load variable, and the functions $\mathit{progress}_Q$ and $\mathit{progress}_A$ reflecting the progress of the locking procedures.

Function *potvec* has been designed such that with every step of $\mathcal{A}_P$, at least one component of the potential vector decreases. If components of *potvec* increase, then there is at least one decreasing component with a smaller index. Hence, with every step of $\mathcal{A}_P$, *potvec* decreases with respect to the lexicographical order. To obtain an upper bound on the move-complexity of $\mathcal{A}_P$ under the central scheduler, we construct a potential function with a scalar value using a weighted sum of the components of *potvec*:

$$\mathit{pot}(c) := \sum_{i=1}^{6} w_i \cdot \mathit{potvec}_i(c)$$

where $w_6 = 1$. The other weights are determined successively from w_5 to w_1 as follows: In order to determine w_j, $1 \leq j \leq 5$, we consider all transitions $c_0 \to c_1$ which are the result of a single move of a single instance and for which potvec_j is the left-most component that changes. Let d_j denote the minimal amount by which potvec_j decreases. Then chose w_j such that the following holds for all such transitions $c_0 \to c_1$:

$$d_j \cdot w_j \geq 1 + \sum_{i=j+1}^{6} w_i(\mathit{potvec}_i(c_1) - \mathit{potvec}_i(c_0))$$

If necessary, w_j is modified such that $w_j \geq 1$ for all $\Delta \geq 1$ and $k \geq 1$. Using this technique, we obtained the values as given below. The reasoning for obtaining each individual weight is contained in the proof of Lemma 6.3.

$$w_1 = 48k\Delta^2 + 16k\Delta - 8k + 13 \qquad w_3 = 12\Delta + 7 \qquad w_5 = 4$$
$$w_2 = 24\Delta^2 + 2\Delta - 2 \qquad w_4 = 12\Delta + 1 \qquad w_6 = 1$$

Lemma 6.3. *Under the central scheduler, pot decreases with every step of* $\mathcal{A}_P$.

Proof. Let $(v, p) \in \mathcal{I}_{\mathcal{A}_P}$ denote an enabled instance of $\mathcal{A}_P$ and let r denote the rule of (v, p) that is enabled We proceed to show that $pot(c)$ decreases under an execution of Rule r of (v, p). A case-by-case analysis is conducted.

$p = P_L$: The number of nodes with an invalid pointer as well as $sqsum(c)$ do not change. Also, $progress_A$ will not increase for any node, since A_a and A_r do not depend on the value of load variables. The number of malicious nodes may decrease, but it cannot grow. The number of nodes with an invalid load variable will decrease by at least 1. $progress_Q$ may increase from 0 to 3 for up to Δ neighbors of v since Q_r or Q_a change for them. For all other nodes, $progress_Q$ does not change. Hence, *pot* decreases by at least $w_4 - 3\Delta \cdot w_5 = 1$.

$p \in \{P_{A_r}, P_{A_a}\}$: (v, P_{A_r}) and (v, P_{A_a}) do not modify $v.\beta$. Hence the number of nodes with an invalid placement does not increase. The number of malicious nodes does not increase since the rule is only enabled if $v.l = L(v)$. Also, $sqsum(c)$ does not change. The number of nodes with an invalid load variable does not increase either. $progress_Q$ does not change for any node. $progress_A$ remains unchanged for all nodes except v, for which it decreases by 1. So *pot* decreases by at least w_6.

$p = P_Q \wedge r =$ P1: The number of nodes with an invalid placement decreases by 1. The number of malicious nodes can increase by at most k. $sqsum(c)$ increases by at most $k(2\Delta - 1)$. The number of nodes with an invalid load variable increases by at most k. $progress_Q$ does not change for any node except v, for which it may increase from 0 to 3. $progress_A$ does not change for any node. Hence, *pot* decreases by at least $w_1 - k \cdot w_2 - k(2\Delta - 1) \cdot w_3 - k \cdot w_4 - 3 \cdot w_5 = 1$.

$p = P_Q \wedge r =$ P2: $progress_Q(v)$ is either 3 or 2 before the move of v. After executing Rule P2, $progress_Q(v)$ is equal to 1. So $progress_Q(v)$ decreases by at least 1. Rule P2 resets $v.q_r$ and $v.q_a$. In the worst case, it holds $v.q_r \in N(v)$ both before and after the move. So $progress_A$ increases by 1 for two neighbors of v. Also, $v.q_a \in N(v)$ might hold prior to the reset of

$v.q_a$ to $\perp$. Hence, $progress_A$ increases by 1 for a third node in the worst case. For all other nodes, $progress_A$ remains unchanged. The number of nodes with invalid placement or load variable does not increase. $sqsum(c)$ does not change. Also, the number of malicious nodes cannot increase, since $Q'_r(v)$ only contains nodes u with $u.a_r \neq v$. So *pot* decreases by at least $w_5 - 3 \cdot w_6 = 1$.

$p = P_Q \wedge r =$ P3: $progress_Q(v)$ is 3 if Rule P3 is enabled. After the execution of Rule P2, $progress_Q(v)$ is either 2 or 1. Hence $progress_Q(v)$ decreases by at least 1. $progress_A$ increases by 1 for at most one node. For all other nodes, $progress_A$ remains unchanged. All other components of the potential function remain unchanged. So *pot* decreases by at least $w_5 - w_6 = 3$

$p = P_Q \wedge r =$ P4: If Rule P4 is enabled, then $progress_Q(v)$ is either 3 or 1. If 3, then it decreases to either 1 or 0. If 1, then it decreases to 0. So $progress_Q(v)$ decreases by at least 1. $progress_A$ increases by 1 for at most one node. For all other nodes, $progress_A$ remains unchanged. All other components of the potential function remain unchanged. So *pot* decreases by at least $w_5 - w_6 = 3$.

$p = P_Q \wedge r =$ P5 $\wedge \neg malicious(v)$: From $\neg malicious(v)$ it follows that $L(v.q_r) > (v.q_r).l$ and $L(v.q_a) < (v.q_a).l$. From the guard of Rule P5 it follows that $v.q_r \in Q_r(v)$ and $v.q_a \in Q_a(v)$ yielding $(v.q_r).l > (v.q_a).l + 1$ and $L(v.q_r) > L(v.q_a)+1$. Hence the $sqsum(c)$ decreases by at least 2. Also, the number of nodes with invalid load variables increases by 2. $progress_Q$ does increase for v from 0 to 2. For all other nodes $progress_Q$ remains unchanged, since $v.l$ is not updated. However, since $v.q_r$ and $v.q_a$ are set to $\perp$, $progress_A$ increases by 1 for two neighbors of v. For all other nodes, $progress_A$ remains unchanged. Also, the number of malicious nodes and the number of nodes with invalid placement does not increase. Hence, *pot* decreases by at least $2 \cdot w_3 - 2 \cdot w_4 - 2 \cdot w_5 - 2 \cdot w_6 = 2$.

$p = P_Q \wedge r =$ P5 $\wedge$ $malicious(v)$: The number of nodes with invalid placement does not increase. However, the number of malicious nodes decreases by at least 1, since both $v.q_r$ and $v.q_a$ are reset to $\perp$. $sqsum(c)$ increases by $2(L(v.q_a) + 1 - L(v.q_r)) \leq 2\Delta - 2$ since $L(v.q_r) \geq 1$ and $L(v.q_a) \leq \Delta - 1$. The number of invalid load variables grows by at most 1, since one load variable must already be outdated for v to be malicious. $progress_Q(v)$ increases from 0 to 2 and $progress_A$ increases by 1 for two nodes. For all other nodes, $progress_Q$ and $progress_A$ remain unchanged. Hence, *pot* decreases by at least $w_2 - (2\Delta - 2) \cdot w_3 - 1 \cdot w_4 - 2 \cdot w_5 - 2 \cdot w_6 = 1$. □

6.5.2 Distributed Scheduler

In the previous section, a potential function for the central scheduler was presented. The distributed scheduler selects a set $S \subseteq \mathcal{I}_{\mathcal{A}_P}$ of enabled instances in each step. Due to the parallel execution of moves, a single step under the distributed scheduler can result in an increase of *pot* when using the weights as defined in Section 6.5.1. This section shows that *pot* is strictly decreasing and thus is a valid potential function under the distributed scheduler if the weights w_3, w_2, and w_1 are increased. The value of *pot* will decrease by at least $|S|$ for every step S under the distributed scheduler. Thus, *pot* is suitable for obtaining an upper bound on the move complexity of $\mathcal{A}_P$ under the distributed scheduler.

For algorithm $\mathcal{A}_P$ proper serializations do not always exist. However, in this section a partial serialization of a step $S \subseteq \mathcal{I}_{\mathcal{A}_P}$ under the distributed scheduler is constructed. It consists of the sequence

$$\langle m_1, m_2, \ldots, m_x, S', m_{x+1}, m_{x+2}, \ldots, m_y \rangle$$

where each $m_i \in S$ denotes a step under the central scheduler and $S' \subseteq S$ denotes as step under the distributed scheduler such that

$$(c : m_1 : m_2 : \ldots : m_x : S' : m_{x+1} : m_{x+2} : \ldots : m_y) = (c : S)$$

The intention is that S' is less complex than S. The partial serialization for a step $S \subseteq \mathcal{I}_{\mathcal{A}_P}$ is constructed via the ranking r. It assigns an element of the lexicographically sorted $\mathbb{Z}^2$ to each enabled instance. Within the serialization, the $m_i \in S$ with $1 \leq i \leq y$ are sorted by their rank $r(m_i)$ in ascending order. The set S' consists of all instances $m_i \in S$ with rank $r(m_i) = (3, 0)$. The instances m_i with $1 \leq i \leq x$ all have a rank less than $(3, 0)$ and the instances m_i with $x < i \leq y$ all have a rank larger than $(3, 0)$.

$$r(v,p) := \begin{cases} (0, 1 - 2v.l) & \text{if } p = P_{A_a} \\ (0, \min\{-2u.l \mid u \in Q_a(v) \wedge u.a_a = v\}) & \text{if } e_Q(4) \\ (1, 0) & \text{if } e_Q(3) \\ (2, 1 + 2v.l) & \text{if } p = P_{A_r} \\ (2, \min\{2u.l \mid u \in Q_r(v) \wedge u.a_r = v\}) & \text{if } e_Q(2) \\ (3, 0) & \text{if } p = P_L \vee e_Q(5) \\ (4, 0) & \text{if } e_Q(1) \\ \bot & \text{otherwise} \end{cases}$$

where $\min \emptyset = \infty$ and the Boolean predicate $e_Q(i)$ is true if and only if $p = P_Q$ and (v, P_Q) is enabled with respect to the i-th rule.

For certain cases, it is not possible to serialize the moves of protocol P_L and instances of P_Q that are enabled with respect to Rule P5. Consider a malicious node $v \in V$. Clearly, $v.q_r \in Q_r(c)$ if (v, P_Q) is enabled with respect to Rule P5. If $(v.q_r, P_L)$ updates the load variables, then the query $v.q_r$ might become invalid as $v.q_r \notin Q_r(v)$ after the move of $(v.q_r, P_L)$. Hence, (v, P_Q) can become disabled with respect to Rule P5 due to the move of $(v.q_r, P_L)$. However, letting (v, P_Q) make a move before $(v.q_r, P_L)$ will change the value of $L(v.q_r)$ and thus the outcome of a move by $(v.q_r, P_L)$. A serialization of other cases might be possible, but was not attempted.

Lemma 6.4. *The ranking r is a weak invariancy ranking. For instances of ranks other than $(3, 0)$, r is a proper invariancy ranking.*

Proof. Let c denote a configuration and $m_1 = (v_1, p_1)$ and $m_2 = (v_2, p_2) \neq m_1$ instances enabled in c such that $r_1 \leq r_2$ where $r_1 = r(m_1)$ and $r_2 = r(m_2)$ with respect to c. By Observation 4.25, assume that $v_1 \in N[v_2]$. It is shown that executing a move of m_1 does not change the rank of m_2 or the outcome of an execution of m_2. If a non-deterministic choice is involved in the execution of m_2, then it will be shown that $Q'_a(v_2)$, or $Q'_r(v_2)$ do not change and that $A_a(v_2)$, $A_r(v_2)$ do not shrink under the execution of m_1, meaning that the element chosen by m_2 is still available after executing m_1.

Assume $x, y \in \mathbb{N}_0$. Let $e_1^Q(i)$ (resp. $e_2^Q(i)$) denote the Boolean predicate which is true if and only if (v_1, P_Q) (resp. (v_2, P_Q)) is enabled with respect to its i-th rule. We conduct a case-by-case analysis.

$r_1 = (0, 1-2x), r_2 = (0, 1-2y)$: It follows that $p_1 = p_2 = P_{A_a}$. m_1 modifies $v_1.a_a$ which does not affect the the rank or result of m_2. In particular $A_a(v_2)$ does not change by prior execution of m_1.

$r_1 = (0, 1-2x), r_2 = (0, -2y)$: It follows that $p_1 = P_{A_a}$, $p_2 = P_Q$, and $e_2^Q(4)$. From $r_1 \leq r_2$ it follows that $x > y$. m_1 changes $v_1.a_a$ when executed. It does not hold $v_1.a_a = v_2$ in c since that would imply $y \geq x$ by definition of $r(m_2)$. Hence, $Q'_a(v_2)$ does not grow if $v_1.a_a$ changes. Also, m_1 would only change $v_1.a_a$ to point towards v_2 if $v_2.q_a = v_1$. In that case, v_2 being enabled with respect to Rule P4 implies $v_1 \notin Q_a(v_2)$. Hence, $Q'_a(v_2)$ does not shrink if $v_1.a_a$ is set to v_2. All other changes of $v_1.a_a$ do not influence $Q'_a(v_2)$ in any way and $\mathit{updateQuery}_a(v_2)$ remains unchanged. Also, $\mathit{queryValidAndAcked}_r(v_2)$ remains unchanged since it depends on q_r and a_r variables which are not changes by m_1.

$r_1 = (0, 1-2x), r_2 = (1, 0)$: It follows that $p_1 = P_{A_a}$, $p_2 = P_Q$, and $e_2^Q(3)$. m_1 changes only $v_1.a_a$ which does not affect the rank or outcome of m_2.

$r_1 = (0, 1-2x), r_2 = (2, 1+2y)$: It follows that $p_1 = P_{A_a}$ and $p_2 = P_{A_r}$. m_1 changes only $v_1.a_a$ which does not affect the rank or outcome of m_2.

$r_1 = (0, 1-2x), r_2 = (2, 2y)$: It follows that $p_1 = P_{A_a}$, $p_2 = P_Q$, and $e_2^Q(2)$. m_1 changes only $v_1.a_a$ which does not affect the rank or outcome of m_2. In particular $Q'_r(v_2)$ does not change by prior execution of m_1.

$r_1 = (0, 1-2x), r_2 = (3, 0), p_2 = P_Q$: It follows that $p_1 = P_{A_a}$ and $e_2^Q(5)$. Thus $queryValidAndAcked_a(v_2)$ is satisfied. If $v_2.q_a = v_1$ then this would imply $v_1.a_a = v_2$. This is a contradiction to the fact that (v_1, P_{A_a}) is enabled. Hence, $v_2.q_a \neq v_1$ and the change of $v_1.a_a$ by m_1 does not impact m_2.

$r_1 = (0, -2x), r_2 = (0, 1-2y)$: It follows that $p_1 = P_Q$, $e_1^Q(4)$, and $p_2 = P_{A_a}$. From $r_1 \leq r_2$ it follows that $x \geq y$. Assume that $A_a(v_2)$ shrinks under the execution of m_1. That implies $v_1.q_a = v_2$ in c. From $e_1^Q(4)$ it follows that $v_2 \notin Q_a(v_1)$. Since $Q_a(v_1)$ includes nodes with minimal load only it must hold that $v_2.l = y > x$. This is a contradiction and hence $A_a(v_2)$ cannot shrink under the execution of m_1.

$r_1 = (0, -2x), r_2 = (0, -2y)$: It follows that $p_1 = p_2 = P_Q$, $v_1 \neq v_2$, $e_1^Q(4)$, and $e_2^Q(4)$. m_1 modifies $v_1.q_a$ which does not affect the rank or result of m_2. In particular $Q'_a(v_2)$ does not change by prior execution of m_1.

$r_1 = (0, -2x), r_2 = (1, 0)$: It follows that $p_1 = p_2 = P_Q$, $v_1 \neq v_2$, $e_1^Q(4)$, and $e_2^Q(3)$. m_1 modifies $v_1.q_a$ which does not affect the rank or result of m_2.

$r_1 = (0, -2x), r_2 = (2, 1+2y)$: It follows that $p_1 = P_Q$, $e_1^Q(4)$, and $p_2 = P_{A_r}$. m_1 modifies $v_1.q_a$ which does not affect the rank or result of m_2.

$r_1 = (0, -2x), r_2 = (2, 2y)$: It follows that $p_1 = p_2 = P_Q$, $v_1 \neq v_2$, $e_1^Q(4)$, and $e_2^Q(2)$. m_1 modifies $v_1.q_a$ which does not affect the rank or result of m_2.

$r_1 = (0, -2x), r_2 = (3, 0), p_2 = P_Q$: It follows that $p_1 = P_Q$, $v_1 \neq v_2$, $e_1^Q(4)$, and $e_2^Q(5)$. m_1 modifies $v_1.q_a$ which does not affect the rank or result of m_2.

$r_1 = (1, 0), r_2 = (1, 0)$: It follows that $p_1 = p_2 = P_Q$, $v_1 \neq v_2$, $e_1^Q(3)$, and $e_2^Q(3)$. m_1 sets $v_1.q_a$ to $\perp$ which does not affect the rank or result of m_2.

$r_1 = (1, 0), r_2 = (2, 1+2y)$: It follows that $p_1 = P_Q$, $e_1^Q(3)$, and $p_2 = P_{A_r}$. m_1 sets $v_1.q_a$ to $\perp$ which does not affect the rank or result of m_2.

$r_1 = (1, 0), r_2 = (2, 2y)$: It follows that $p_1 = p_2 = P_Q$, $v_1 \neq v_2$, $e_1^Q(3)$, and $e_2^Q(2)$. m_1 sets $v_1.q_a$ to $\perp$ which does not affect the rank or result of m_2.

$r_1 = (1,0), r_2 = (3,0), p_2 = P_Q$: It follows that $p_1 = P_Q$, $v_1 \neq v_2$, $e_1^Q(3)$, and $e_2^Q(5)$. m_1 sets $v_1.q_a$ to $\perp$ which does not affect the rank or result of m_2.

$r_1 = (2, 1+2x), r_2 = (2, 1+2y)$: It follows that $p_1 = p_2 = P_{A_r}$. m_1 modifies $v_1.a_r$ which does not affect the the rank or result of m_2. In particular $A_r(v_2)$ does not change by prior execution of m_1.

$r_1 = (2, 1+2x), r_2 = (2, 2y)$: It follows that $p_1 = P_{A_r}$, $p_2 = P_Q$, and $e_2^Q(2)$. From $r_1 \leq r_2$ it follows that $x < y$. m_1 changes $v_1.a_r$ when executed. It does not hold $v_1.a_r = v_2$ in c since that would imply $y \leq x$ by definition of $r(m_2)$. Hence, $Q'_r(v_2)$ cannot grow if $v_1.a_r$ changes. Also, m_1 would only change $v_1.a_r$ to point towards v_2 if $v_2.q_r = v_1$. In that case, v_2 being enabled with respect to Rule P2 implies $v_1 \notin Q_r(v_2)$. Hence, $Q'_r(v_2)$ does not shrink if $v_1.a_r$ is set to v_2. All other changes of $v_1.a_r$ do not influence $Q'_r(v_2)$ in any way and $\mathit{updateQuery}_r(v_2)$ remains unchanged.

$r_1 = (2, 1+2x), r_2 = (3,0), p_2 = P_Q$: It follows that $p_1 = P_{A_r}$ and $e_2^Q(5)$. Thus $\mathit{queryValidAndAcked}_r(v_2)$ is satisfied. If $v_2.q_r = v_1$ then this would imply $v_1.a_r = v_2$. This is a contradiction to the fact that (v_1, P_{A_r}) is enabled. Hence, $v_2.q_r \neq v_1$ and the change of $v_1.a_r$ by m_1 does not impact m_2.

$r_1 = (2, 2x), r_2 = (2, 1+2y)$: It follows that $p_1 = P_Q$, $e_1^Q(2)$, and $p_2 = P_{A_r}$. From $r_1 \leq r_2$ it follows that $x \leq y$. Assume that $A_r(v_2)$ shrinks under the execution of m_1. That implies $v_1.q_r = v_2$ in c. From $e_1^Q(2)$ it follows that $v_2 \notin Q_r(v_1)$. Since $Q_r(v_1)$ includes nodes with maximal load only it must hold that $v_2.l = y < x$. This is a contradiction and hence $A_r(v_2)$ cannot shrink under execution of m_1.

$r_1 = (2, 2x), r_2 = (2, 2y)$: It follows that $p_1 = p_2 = P_Q$, $v_1 \neq v_2$, $e_1^Q(2)$, and $e_2^Q(2)$. m_1 modifies only $v_1.q_r$ and $v_1.q_a$ which does not affect the rank or result of m_2. In particular $Q'_r(v_2)$ does not change by prior execution of m_1.

$r_1 = (2, 2x), r_2 = (3,0), p_2 = P_Q$: It follows that $p_1 = P_Q$, $v_1 \neq v_2$, $e_1^Q(2)$, and $e_2^Q(5)$. m_1 modifies only $v_1.q_r$ and $v_1.q_a$ which does not affect the rank or result of m_2.

$r_1 < r_2 = (3,0), p_2 = P_L$. It follows that m_1 does not change $v_1.\beta$. Hence, $L(v_2)$ does not change.

$r_1 = (3,0), r_2 = (3,0)$: m_1 and m_2 are in parallel as part of a step under the distributed scheduler.

$r_1 \leq r_2 = (4,0)$: It follows that $p_2 = P_Q$, and $e_2^Q(1)$. Furthermore, either $p_1 \neq P_Q$ or $v_1 \neq v_2$. Hence, m_1 does not modify $v_2.\beta$ and therefore $\mathit{validPlacement}(v_2)$ does not change. □

Lemma 6.5. *The potential function pot decreases by at least $|S|$ with every step $S \subseteq \mathcal{I}_{\mathcal{A}_P}$ of $\mathcal{A}_P$ under the distributed scheduler if*

$$w_1 = 96k\Delta^2 + 60k\Delta - 22k + 13 \qquad w_3 = 24\Delta + 21 \qquad w_5 = 4$$
$$w_2 = 48\Delta^2 + 30\Delta - 2 \qquad w_4 = 12\Delta + 1 \qquad w_6 = 1$$

Proof. Let $S \subseteq \mathcal{I}_{\mathcal{A}_P}$ be a step under the distributed scheduler. Lemma 6.4 shows that $\langle m_1, m_2, \ldots, m_x, S', m_{x+1}, m_{x+2}, \ldots, m_y \rangle$ is a valid partial serialization. Let c_0 denote the configuration prior to the step S. Let c_1 denote configuration $(c : m_1 : m_2 : \ldots : m_x)$, c_3 configuration $(c_1 : S')$, and c_4 configuration $(c_3 : m_{x+1} : m_{x+2} : \ldots : m_y)$. Then $c_4 = (c_0 : S)$ holds. It is easy to verify that Lemma 6.3 still holds for the adjusted values of w_1 to w_3. Lemma 6.3 then yields that $pot(c_1) \leq pot(c_0) - x$ and $pot(c_4) \leq pot(c_3) - (y - x)$. To prove the claim it remains to show that $pot(c_3) \leq pot(c_1) - |S'|$.

S' only contains instances of P_L and instances of P_Q which are enabled with respect to Rule P5. Note that the sequential execution of all (v, P_Q) yields the same result as their parallel execution in a single step. This is due to the fact that Rule P5 only changes $v.\beta$, $v.q_r$, and $v.q_a$, all of which do not impact the execution of Rule P5 by a neighboring instance of P_Q in any way. Hence, S' can be replaced by a sequential execution of Rule P5 for all instances $(v, P_Q) \in S'$ under the central scheduler followed by a sequence of moves that set the load variable $v.l$ to the value of $L(v)$ in c_1 for each $(v, P_L) \in S'$. Let c_2 denote the configuration after all executions of Rule P5 and prior to the moves that set the load variables.

Note that *pot* decreases from c_1 to c_2 by Lemma 6.3. However, from c_2 to c_3, the value of *pot* may increase due to the moves which change load variables. We distinguish three types of nodes $v \in V$:

Type A: The value of $L(v)$ is identical in c_1 and c_2.

Type B: The value of $L(v)$ differs in c_1 and c_2 and $L(v) \neq v.l$ in c_2.

Type C: The value of $L(c)$ differs in c_1 and c_2 and $L(v) = v.l$ holds in c_2.

If v is of type A then the update of $v.l$ is identical to a regular move of (v, P_L) and *pot* decreases by Lemma 6.3. If v is of type B, the number of nodes with an invalid load variable does not increase. However, replacing one invalid counter value with another may invalidate queries of at most two $u \in N(v)$. For them, $progress_Q(u)$ increases by at most 3. If v is of type C, $\# v \in V : \neg validLoad(v)$ increases by one in addition to the increase of $progress_Q$ as described for type B. In total, the update of $v.l$ increases *pot* by at most $1 \cdot w_4 + 3 \cdot w_5 = 12\Delta + 13$.

For all three types, neither $progress_A$ nor *validPlacement* change for any node. Also, the number of malicious nodes does not increase from c_2 to c_3. Assume $\neg malicious(v)$ in c_2. If $(v, P_Q) \in S'$ then $v.q_r = v.q_a = \bot$ in c_2 and thus in c_3. Hence v is not malicious in c_3. If $(v, P_Q) \notin S'$, v and $(u, P_L) \in S'$ with $v.q_r = u$ and $u.a_r = v$, then $L(u) \geq u.l$ in c_2. From $u.a_r = v$ it follows that $L(u)$ cannot decrease from c_1 to c_2. Hence, $L(u) \geq u.l$ still holds in c_3 and v is not malicious in c_3. The case $v.q_a = u$ and $u.a_a = v$ is analogous.

An execution of Rule P5 by each $(v, P_Q) \in S'$ results in at most two type B/C nodes (namely $v.q_r$ and $v.q_a$). All other nodes are of type A. We now show that each $(v, P_Q) \in S'$ compensates the increase of *pot* for at least 2 type C nodes. Following the same reasoning as in the proof of Lemma 6.3, a move (v, P_Q) decreases *pot* by at least $2 \cdot w_3 - 2 \cdot w_4 - 2 \cdot w_5 - 2 \cdot w_6 = 24\Delta + 30$ if $\neg malicious(v)$ and $w_2 - (2\Delta - 2) \cdot w_3 - 1 \cdot w_4 - 2 \cdot w_5 - 2 \cdot w_6 = 24\Delta + 29$ otherwise. It follows that $pot(c_3) \leq pot(c_1) - |S'|$. □

6.6 Analysis

Lemma 6.6 (Partial Correctness)**.** *If $\mathcal{A}_P$ has terminated, then $\mathcal{L}_{\mathcal{A}_P}$ is satisfied.*

Proof. $validLoad(v)$, $validPlacement(v)$, and $\neg updateQuery_r(v)$ hold for all nodes $v \in V$ (Protocol P_L and Rules P1 and P2).

Assume there exists a node $v \in V$ such that $w = v.a_a \neq \bot$. Protocol P_{A_a} implies that $w \in A_a(v)$, hence $w.q_a = v$ and $(w.q_a).a_a = w$. Rule P3 implies $queryValidAndAcked_r(w)$. This yields that $w.q_a \in Q_a(w)$ (Rule P4). Furthermore, by Rule P5, $queryValidAndAcked_a(w)$ is false, hence $(w.q_a).a_a \neq w$. This contradiction proves that all $v \in V$ satisfy $v.a_a = \bot$ and $A_a(v) = \emptyset$.

Assume there exists $v \in V$ such that $w = v.q_a \neq \bot$. Then $w \in v.\beta$ since $A_a(w) = \emptyset$. This implies $w \notin Q_a(v)$ and hence $updateQuery_a(v)$ is true. By Rule P4 $queryValidAndAcked_r(w)$ is false, therefore $v.q_a = \bot$ by Rule P3. This contradiction proves that all $v \in V$ satisfy $v.q_a = \bot$.

Assume there exists $v \in V$ such that $w = v.q_r \neq \bot$. $\neg updateQuery_r(v)$ implies $v.q_r \in Q_r(v)$, hence $Q_a(v) \neq \emptyset$. Protocol P_{A_r} and $A_r(w) \neq \emptyset$ imply that $w.a_r \neq \bot$. W.l.o.g. assume $v = w.a_r$. Hence $queryValidAndAcked_r(v)$ and thus $v.q_a \neq \bot$ by Rule P4. This contradicts the result of the last paragraph. Hence $v.q_r = \bot$ for all $v \in V$. This yields $A_r(v) = \emptyset$ for all $v \in V$ and thus $v.a_r = \bot$ by Protocol P_{A_r}. Furthermore, $Q_r(v) = Q'_r(v)$ for all $v \in V$. Thus, $Q_r(v) = \emptyset$ by Rule P2. This proves that $\mathcal{L}_{\mathcal{A}_P}$ is satisfied. □

Lemma 6.7 (Closure)**.** *If $\mathcal{L}_{\mathcal{A}_P}$ is satisfied, then $\mathcal{A}_P$ has terminated.*

Proof. Let v denote a node and consider a legitimate configuration. (v, P_L) and Rule P1 are disabled for v since *validLoad*(v) and *validPlacement*(v). It also holds $v.a_r = v.a_a = \perp$ and $u.q_r = u.q_a = \perp$ for all neighbors $u \in N(v)$. Hence, (v, P_{A_r}) and (v, P_{A_a}) are disabled for v since $A_r(v) = A_a(v) = \emptyset$. From $Q_r(v) = \emptyset$ it follows that $Q'_r(v) = \emptyset$. Since $v.q_r = \perp$, Rule P2 is disabled. Rule P3 is disabled since $v.q_a = \perp$. Rules P4 and P5 are disabled since the query $v.q_r = \perp$ is not valid. □

Theorem 6.8. *Algorithm $\mathcal{A}_P$ is self-stabilizing under the distributed scheduler*

Proof. Correctness and closure follow from Lemmas 6.6 and 6.7 and termination from Lemmas 6.3 and 6.5. □

Theorem 6.9. *In a legitimate configuration, the variables $v.\beta$ of all $v \in V$ induce a local k-placement with local minimum variance.*

Proof. For each $v \in V$ let $\beta(v) = v.\beta$. $\mathcal{L}_{\mathcal{A}_P}$ implies *validPlacement*(v) and therefore $\beta(v) \subseteq N(v)$ and $|\beta(v)| = k(v)$ for all $v \in V$. Hence, β is a local k-placement. From $Q_r(v) = \emptyset$ for all $v \in V$ it follows that no node can lower the variance by moving a replica from one of its neighbors to another. □

Theorem 6.10. *Algorithm $\mathcal{A}_P$ requires $\mathcal{O}(\log n)$ bits per node.*

Proof. For each node $v \in V$, $v.\beta$ required $\mathcal{O}(k \log n)$ bits. The four variables $v.q_r$, $v.q_a$, $v.a_r$, and $v.a_a$ require $\mathcal{O}(\log n)$ bits each. □

Theorem 6.11. *Algorithm $\mathcal{A}_P$ terminates after at most $\mathcal{O}(n\Delta^2)$ moves under the distributed scheduler.*

Proof. For each node $v \in V$, $L(v)$ can not be larger than Δ. The value of *sqsum* reaches its maximum if the $k \cdot n$ replicas are concentrated on as few nodes as possible. Hence, *sqsum*(c) can not become larger than $\lceil \frac{kn}{\Delta} \rceil \Delta^2 \in \mathcal{O}(n\Delta)$ and $w_3 sqsum(c) \in \mathcal{O}(n\Delta^2)$. All other components of *potvec* are $\mathcal{O}(n)$ and the w_j are $\mathcal{O}(\Delta^2)$. Hence $pot(c) \in \mathcal{O}(n\Delta^2)$. The claim follows from Lemmas 6.3 and 6.5. □

Theorem 6.12. *Algorithm $\mathcal{A}_P$ terminates after at most $\mathcal{O}(n\Delta)$ rounds under the distributed scheduler.*

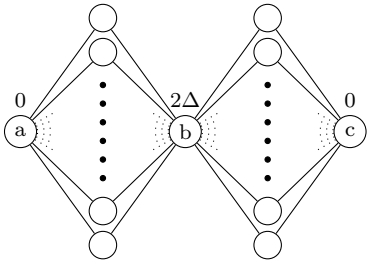
Proof. Let c_0 be the initial configuration of an execution and c_i denote the configuration at the end of round i. Note that c_{i-1} is the first configuration of round i. Within the first round, each node that is enabled with respect to Rule P1 in c_0 executes Rule P1. Note that $\#\, v \in V : \mathit{validPlacement}(v)$ never increases. Hence in c_1 all nodes $v \in V$ satisfy $\mathit{validPlacement}(v)$ and any execution starting in a configuration such as c_1 never contains any moves of Rule P1.

During the second round, all malicious nodes become non-malicious either by executing Rule P5 or due to a move of Protocol P_L of neighboring nodes. Hence in c_2, all nodes $v \in V$ satisfy $\neg\mathit{malicious}(v)$ and $\mathit{validPlacement}(v)$.

Let c_2 denote any configuration in which all nodes satisfy $\neg\mathit{malicious}(v)$ and $\mathit{validPlacement}(v)$. It is now shown that any execution e starting in c_2 that does not contain a move of Rule P5 is at most 5 rounds in length. Since e does not contain an execution of Rule P5, it must hold that in c_3 and all subsequent configurations of e, all nodes $v \in V$ satisfy $\mathit{validLoad}(v)$. Hence, $Q_r(v)$ and $Q_a(v)$ are constant in the suffix of e starting with c_3. Let $v \in V$ with $Q_r(v) \neq \emptyset$ and let $v.q_r$ and $v.q_a$ be invalid or equal to $\perp$. In c_4, $v.q_r$ is valid. In c_5, the node $u = v.q_r$ acknowledges a removal query. W.l.o.g. let $u.a_r = v$. In c_6, $v.q_a$ is a valid query. In c_7, the node $w = v.q_a$ acknowledges an additional query. W.l.o.g. let $w.a_a = v$. Starting with c_7, v is enabled with respect to Rule P5. If the execution would continue for another round, then v would execute Rule P5. If $Q_r(v) = \emptyset$ for all nodes $v \in V$ in c_2, then all nodes reset their queries to $\perp$ and all nodes have updated their load variable in c_3. In c_4, all nodes have reset their acknowledgments to $\perp$. The algorithm then terminates in c_4. The claim follows from the fact that the maximum of *sqsum* is $\mathcal{O}(n\Delta)$ (cf. the proof of Theorem 6.11). □

We are aware of scenarios, in which algorithm $\mathcal{A}_P$ requires $\mathcal{O}(\Delta)$ rounds. In particular, for a complete graph, $\mathcal{A}_P$ requires $\mathcal{O}(n) = \mathcal{O}(\Delta)$ rounds. We are not aware of an example where algorithm $\mathcal{A}_P$ actually requires $\mathcal{O}(n\Delta)$ rounds. The proof of Theorem 6.12 does not consider that multiple nodes can move their replicas simultaneously. However, how many nodes can do so heavily depends on the topology and the initial local k-placement. It seems difficult to account for this in the analysis.

For the complete graph the number of edges is in $\Omega(\Delta^2)$. We now present an example where this number is in $\Omega(\Delta)$ and $\mathcal{A}_P$ still requires $\Omega(\Delta)$ rounds to terminate. Consider the topology shown in Figure 6.6. Assume $k = 1$. We only focus on the load of nodes a, b, and c. The values of $a.\beta$, $b.\beta$, and $c.\beta$ settle within a few rounds. Initially, the replicas of all neighbors of b are placed on b. Then $L(a) = L(c) = 0$ and $L(b) = 2\Delta$. Let the scheduler first



$L(a)$	$L(b)$	$L(c)$	Replicas moved
0	2Δ	0	Δ from b to a
Δ	Δ	0	$\frac{\Delta}{2}$ from b to c
Δ	$\frac{\Delta}{2}$	$\frac{\Delta}{2}$	$\frac{\Delta}{4}$ from a to b
$\frac{3\Delta}{4}$	$\frac{3\Delta}{4}$	$\frac{\Delta}{2}$	$\frac{\Delta}{8}$ from b to c
$\frac{3\Delta}{4}$	$\frac{5\Delta}{8}$	$\frac{5\Delta}{8}$	$\frac{\Delta}{16}$ from a to b
$\frac{11\Delta}{16}$	$\frac{11\Delta}{16}$	$\frac{5\Delta}{8}$	$\frac{\Delta}{32}$ from b to c
$\frac{11\Delta}{16}$	$\frac{21\Delta}{32}$	$\frac{21\Delta}{32}$	...

Figure 6.6: Example topology with $\Delta = \deg(a) = \deg(b) = \deg(c)$, $L(a) = L(c) = 0$, and $L(b) = 2\Delta$ (left) and possible execution of $\mathcal{A}_P$ (right)

select all neighbors of b. They query node b for removal. It is assumed that for the next Δ acknowledgments by node b, the non-deterministic choice of b selects common neighbors of b and a. Hence, Δ-many common neighbors of a and b are able to move Δ replicas from b to a. This takes 3 rounds per replica, one round for each query to a, the acknowledgment of a, and the adjustment of the replica placement. Hence, at least 3Δ rounds pass until the load of a is Δ. Afterwards, it holds $L(a) = L(b) = \Delta$ and $L(c) = 0$ remains unchanged.

Next, $\frac{\Delta}{2}$ replicas are moved from b to c, $\frac{\Delta}{4}$ replicas are moved from a to b, and so forth, as shown in the table in Figure 6.6. This continues until nodes a, b, and c all have load $\frac{\Delta}{3}$. In total, no more than 2Δ replicas will be moved by the neighbors of b. In the proof of Theorem 6.12, it has been shown that this takes at most 5 rounds per replica. Hence, the execution of $\mathcal{A}_P$ terminates after at most $\mathcal{O}(\Delta)$ rounds.

We are unable to construct an execution that takes more than $\mathcal{O}(\Delta)$ rounds, neither for the topology shown in Figure 6.6 nor any other topology. Nevertheless, we conjecture that $\mathcal{A}$ always stabilizes in $O(n)$ rounds.

6.7 Integration

This section describes the modifications required for $\mathcal{A}_{FL}$ to place the backups according to a local 2-placement computed by $\mathcal{A}_P$. Recall the implementation of $\mathcal{A}_{FL}$ as presented in Section 5.8. Algorithm $\mathcal{A}$ denotes the input to the transformation.

Procedure $action_Q(v)$

Nodes: v is the current node
Variables: $v.p \in \sigma_{\mathcal{A}}$; $v.bptr$: set of at most 2 node identifiers

if $v.q =$ REPAIRED **then**
  if $\exists u \in N(v) : u.b_v = v.p$ **then** Keep value of $v.p$
  else if $\exists u, w \in N(v) : w \neq u \wedge u.b_v = w.b_v \neq \perp$ **then** $v.p := u.b_v$
  else if $\deg(v) = 1$ **then**
    $u :=$ neighbor of node v;
    if $u.d_v =$ UPDATE $\wedge\ u.b_v \neq \perp$ **then** $v.p := u.b_v$
    else if $u.d_v =$ SINGLE **then** $pairRepair(v, u)$
  end
end
if $v.q =$ EXECUTED **then**
  $v.bptr := BP(v)$;
  if $G_{\mathcal{A}}(v)$ **then** $S_{\mathcal{A}}$
end

Procedure $pairRepair(v, u)$

if $v.b_u \neq u.p \wedge v.id < u.id \wedge v.b_u \neq \perp \wedge$
  $\neg pairEnabled_{\mathcal{A}}(v : v.p, u : v.b_u)$ **then** Keep value of $v.p$
else if $pairEnabled_{\mathcal{A}}(u : u.p, v : v.p) \wedge u.b_v \neq \perp \wedge$
  $\neg pairEnabled_{\mathcal{A}}(u : u.p, v : u.b_v)$ **then** $v.p := u.b_v$

Procedure $action_{R_v}(u)$

Nodes: u is the current node, $v \in N(u)$ is the center of the cell
Variables: $u.r_v \in \{$KEEP, UPDATE, SINGLE$\}$; $u.b_v \in \sigma_{\mathcal{A}} \cup \{\perp\}$

if $v.q =$ REPAIRED **then**
  if $\deg(u) = 1$ **then** $u.d_v :=$ SINGLE
  else if $(\exists w \in N(u) : w.b_u = u.p) \wedge$
    $(G_{\mathcal{A}}(u) \vee G_{\mathcal{A}}(v : v.p, u : u.p))$ **then** $u.d_v :=$ UPDATE
  else $u.d_v :=$ KEEP
end
if $v.q =$ PAUSED **then**
  if $u \in v.bptr$ **then** $u.b_v := v.p$ **else** $u.b_v := \perp$
end

Figure 6.7: Revised implementation of transition actions

$$
\begin{aligned}
\mathit{backupConsistent}(v) \equiv\ & (\forall u \in N(v) : u.b_v \in \{\bot, v.p\}) \ \wedge \\
& (N(v) \neq \emptyset \Rightarrow \exists u \in N(v) : u.b_v = v.p) \\
\mathit{repaired}(v) \equiv\ & \mathit{dialogConsistent}(v) \ \wedge \ (\mathit{backupConsistent}(v) \ \vee \\
& v.s = \textsc{Repaired} \ \vee \ (v.s = \textsc{Executed} \ \wedge \\
& (\forall u \in N(v) : u.r_v = \textsc{Paused} \Rightarrow u.b_v \in \{v.p, \bot\}) \ \wedge \\
& (N(v) \neq \emptyset \Rightarrow \exists u \in N(v) : (u \in v.\mathit{bptr} \wedge u.r_v \neq \textsc{Paused}) \ \vee \\
& \qquad (u.r_v = \textsc{Paused} \ \wedge \ u.b_v = v.p))))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{actualBP}(v) &:= \{u \in N(v) \mid u.b_v \neq \bot\} \\
\mathit{startCond}_Q(v) &\equiv G_{\mathcal{A}}(v) \ \vee \ \neg(\mathit{backupConsistent}(v) \ \wedge \\
& \quad \mathit{actualBP}(v) = \mathit{BP}(v))
\end{aligned}
$$

Figure 6.8: Revised predicates

The domain of the backup variable $u.b_v$ of each responding instance (u, R_v) is extended such that it includes a special value $\bot$. This special value denotes that (u, R_v) does not hold a backup of $v.p$. It is assumed that $u.b_v$ is stored in such a way that the special value $\bot$ consumes only a constant number of bits whereas each regular backup $u.b_v \neq \bot$ requires $\mathcal{O}(s_{\mathcal{A}})$ bits, where $s_{\mathcal{A}}$ denotes the number of bits required to store a local state of $\mathcal{A}$. To the local state of each center instance (v, Q), we add the variable $v.\mathit{bptr}$ which is a set of at most two node identifiers. A responding instance (u, R_v) sets its backup to either $v.p$ or $\bot$ depending on whether $u \in v.\mathit{bptr}$ when providing the acknowledgment for a transition to position Paused. The value of $v.\mathit{bptr}$ is updated before (v, Q) makes the query for a transition Paused.

A backup variable $u.b_v$ with $u.b_v = v.p$ and $u \in N(v)$ is called a *confirmation* of $v.p$. We define a function BP which assigns to each node a set of at most two neighbors. If $v.\beta \cap N(v) \neq \emptyset$, then $\mathit{BP}(v)$ returns the value of $v.\beta$. Otherwise, $\mathit{BP}(v)$ deterministically selects one arbitrary neighbor of v. This is necessary to ensure that at least one confirmation of $v.p$ exists if the transition to Paused is completed, so that no repair of the primary variable $v.p$ is performed during the next cycle.

Replacements of procedures action_Q and action_{R_v} that implement the above are provided in Figure 6.7. The repair mechanism implemented in action_Q was revised in order to support the fact that at most two backups exist. Also, a state corruption may delete a backup, i.e., set the backup variable to $\bot$. We proceed to discuss the revised repair mechanism of action_Q. Consider a 1-faulty configuration. Clearly, a primary variable $v.p$ was not

corrupted if a confirmation exists (line 2). Otherwise, if no confirmation exists, but two backups with identical value exist, then $v.p$ is updated (line 3). The repair mechanism for the case where node v only has a single neighbor is not changed and is based on Sections 5.8.4.2 and 5.8.4.3. In line 3 of $action_{R_v}(u)$ however, the condition was revised such that a single confirmation of $u.p$ suffices. Furthermore, procedure *pairRepair* was updated to ignore backup variables that have value $\perp$.

In Figure 6.8, redefinitions of several predicates of $\mathcal{A}_{FL}$ are given. A cell C_v is called backup-consistent if all backups not equal to $\perp$ are confirmations of $v.p$ and if at least one confirmation exists. The predicate $repaired(v)$ has been revised such that it is true for all cells at positions Repaired and Executed that are backup-consistent when they finish the transition to Paused. As discussed above, this is necessary to ensure that no repair of $v.p$ will be performed during a subsequent cycle. Furthermore, the predicate $startCond_Q(v)$ has been adjusted such that a cell starts a new cycle if the backup placement within C_v does not match the one returned by $BP(v)$. At most one cycle is needed to move the backups to the nodes returned by $BP(v)$, assuming that $BP(v)$ does not change.

With these adjustments in place, a new transformation can be created that only uses 2 backups per node and balances the load of the nodes. One way to do this is to use the collateral composition $\mathcal{A}_P \triangleright \mathcal{A}_{FL}$. It can be shown that the adjusted version of $\mathcal{A}_{FL}$ is strongly silent. However, while the containment time remains constant, the fault-gap increases from $\mathcal{O}(1)$ rounds to the number of rounds needed by $\mathcal{A}_P$ to terminate when the initial configuration is 1-faulty. Hence, we propose to use the independent composition $\mathcal{A}_P + \mathcal{A}$ as the input to the transformation $\mathcal{A}_{FL}$, where $\mathcal{A}$ is the algorithm to be transformed. For any 1-faulty configuration $\mathcal{A}_{FL}$ repairs the variables of $\mathcal{A}_P$ and $\mathcal{A}$ within $\mathcal{O}(1)$ rounds. In particular, the repair of $v.\beta$ is complete before the return value of $BP(v)$ is used by $action_Q(v)$.

Let $\mathcal{A}_P + \mathcal{A}$ be the input algorithm of $\mathcal{A}_{FL}$. Then the following theorems hold by Theorem 4.8:

Theorem 6.13 (Fault-Containment)**.** *$\mathcal{A}_{FL}$ is fault-containing with respect to $\mathcal{L}_{\mathcal{A}}$. $\mathcal{A}_{FL}$ has a containment time and fault-gap of $\mathcal{O}(1)$ rounds, a contamination number of 1, and a fault-impact of $\mathcal{O}(\Delta^2)$ (radius 2).*

Theorem 6.14 (Stabilization Time)**.** *Any execution of $\mathcal{A}_{FL}$ terminates after at most $6\max(T_{\mathcal{A}_P}, T_{\mathcal{A}}) + 11$ rounds, where $T_{\mathcal{A}_P}$ and $T_{\mathcal{A}}$ denote the maximum length of any execution of $\mathcal{A}_P$ and $\mathcal{A}$ in rounds respectively.*

Theorem 6.15 (Stabilization Space)**.** *The average space required per node by $\mathcal{A}_{FL}$ in a legitimate configuration is $\mathcal{O}(s_{\mathcal{A}} + \Delta + \log n)$ where $s_{\mathcal{A}}$ is the space required by $\mathcal{A}$ per node in a legitimate configuration.*

Proof. By Theorem 6.10, $\mathcal{A}_P + \mathcal{A}$ requires $s = s_{\mathcal{A}} + \log n$ bits per node. On average, each node stores two backups. Hence, $3s$ bits per node are required on average. The variable $v.bptr$ requires $\mathcal{O}(\log n)$ bits per node. The dialog and decision variables take $\mathcal{O}(\Delta)$ bits per node. □

We do not give formal proofs of the correctness of Theorems 6.13 and 6.14. In Chapter 7, a more complex transformation based on the modifications presented in this section is proven correct. In particular the proofs of termination and slowdown would be based on the following key aspects:

- Slowdown: Lemma 5.27 also holds if the output of $\mathcal{A}_P$ changes during the execution of $\mathcal{A}_{FL}$.
- Termination: The output of $\mathcal{A}_P$ changes at most a finite number of times. Hence, a cell can only execute a finite number of cycles solely for the purpose of adjusting the placement of the backups. Hence, in the proof of Theorem 5.26 (Termination) the number of cycles per cell remains finite.

6.8 Concluding Remarks

This chapter presented a novel self-stabilizing algorithm $\mathcal{A}_P$ for the k-placement problem that stabilizes in $\mathcal{O}(n\Delta^2)$ moves or $\mathcal{O}(n\Delta)$ rounds respectively under the unfair distributed scheduler. It computes k-placements where the standard deviation of the load per node assumes a local minimum.

Another obvious approach for solving the k-placement problem is to design an algorithm in the distance-2 model. Note that under the distributed scheduler, a naive algorithm (merely consisting of the equivalent of Rule P5) would be vulnerable to a livelock resembling the one shown in Figure 6.4. Also, transforming the algorithm to the distance-1 model as described in [Tur12, GGH$^+$04] leads to an increase of the move-complexity by a factor of $\mathcal{O}(m)$. The resulting algorithm would thus not achieve the efficiency of $\mathcal{A}_P$. Furthermore, the transformations assume globally unique node identifiers, while $\mathcal{A}_P$ only required weakly unique identifiers.

As laid out in Section 6.7, $\mathcal{A}_P$ complements the transformation for fault-containment described in Section 5.8. But the algorithm can also be useful outside the context of this transformation.

A more flexible placement of replicas may be of interest. In particular, placement that is not restricted to the 1-hop neighborhood of a node. For a star graph, a local k-placement will always result in load Δ for the center of the star. While placing the replicas at larger distances from each node may allow a more homogeneous distribution of the load per node for certain topologies, it also implies more communication overhead. In the context of self-stabilizing systems, we assume that replicas should be regularly checked for corruption. Hence, communication between the node that stores the replica and the source of the replica is required.

7 Fault-Containment in Dynamic Networks

Dynamic systems form a class of systems where the communication links and nodes may fail and recover at runtime. As discussed in Section 3.3, self-stabilizing algorithms do in principle eventually recover from such faults. But their behavior during that time is undefined. Furthermore, self-stabilizing protocols are often only optimized with respect to their worst-case stabilization time, but not with respect to their behavior in case of small-scale faults. Additional efforts are required to deal with small-scale faults. Techniques that address small-scale state corruption have been presented in Chapter 5. Small-scale topology changes have been addressed by Dolev and Herman [DH95, DH97]. They introduce the concept of super-stabilization. A super-stabilizing algorithm is self-stabilizing but also maintains a safety property subsequent to a topology change, provided that the configuration prior to the topology change was legitimate.

Dolev and Herman [DH97] assume that the distributed system is equipped with a mechanism to detect topology changes. We argue that variables used by this mechanism must be regarded as subject to state corruptions, which may occur along with the topology change. It is an open question whether it is feasible to reliably detect topology changes in spite of state corruptions.

This chapter addresses this issue by combining fault-containment, i.e., the fast repair of state corruptions, with super-stabilization, i.e., maintaining a safety property after a topology change. The aim is to create distributed algorithms that provide fault-containing self-stabilization, super-stabilization, and in addition maintain a safety property if a topology change and a state corruption occur simultaneously. To the best of our knowledge, this work is the first to consider this combination of faults.

The transformations given in Chapter 5 do not solve this problem. They use a specific technique to recover from an insufficient number of backups which breaks in face of topology changes. Ideally, they should gracefully handle the case in which backups become unreachable due to a topology change.

Section 7.1 describes the concept of super-stabilization. Section 7.2 defines the novel concept of fault-containing super-stabilization. The main contribution of this chapter is the transformation presented in Section 7.3.

It is able to add fault-containment to any silent super-stabilizing distributed algorithm. The transformation preserves existing super-stabilization properties of the input algorithm such that the transformed protocol, i.e., the output of the transformation, is super-stabilizing even if an additional state corruption happens simultaneously with the topology change.

The transformation distributes at most 4 backups of each node's local state within the 2-hop neighborhood. It focuses on topology changes that add or remove a single edge. More far-reaching topology changes are discussed in Section 7.6. The given transformation shows that reliably detecting topology changes in spite of state corruption is feasible. The transformation is an enhancement of the transformation given in Section 5.8. Hence, we recommend reading that section prior to proceeding with this chapter.

7.1 Super-Stabilization

This section presents the definition of super-stabilization used in this chapter. It is based on the one provided by Dolev and Herman in [DH95, DH97].

The definition of super-stabilization focuses on the behavior of a self-stabilizing algorithm after a topology change. Topology changes may include edge removal and edge additions. Node crashes or recoveries are modeled as removal and addition of all adjacent edges. Let $\mathcal{A}$ denote a distributed algorithm that is self-stabilizing with respect to the Boolean predicate $\mathcal{L}_\mathcal{A}$ and Λ denote a class of topology changes. To describe the configuration of a system after a topology change of class Λ, the following notion is defined:

Definition 7.1. An extended configuration $(E, c) \in \Sigma_\mathcal{A}^{ext}$ is called Λ-*faulty* if an extended configuration $(E', c) \in \Sigma_\mathcal{A}^{ext}$ exists such that

- (E', c) satisfies $\mathcal{L}_\mathcal{A}$ and
- E differs from E' in a single topology change of class Λ.

For any execution starting in a Λ-faulty configuration, a Λ-super-stabilizing algorithm $\mathcal{A}$ guarantees that after a constant number of rounds, all configurations satisfy the safety property. This safety property is described by a Boolean predicate $\mathcal{S}_\mathcal{A}$. The predicate is usually very application specific. Examples of safety properties are provided later in this section. A configuration that satisfies $\mathcal{S}_\mathcal{A}$ is called *safe*. It is required that every legitimate configuration is also safe, i.e., $\mathcal{L}_\mathcal{A} \Rightarrow \mathcal{S}_\mathcal{A}$ holds for all configurations.

Definition 7.2 (Super-Stabilization)**.** $\mathcal{A}$ is Λ-super-stabilizing with respect to $\mathcal{S}_\mathcal{A}$ if a constant $t \in \mathbb{N}_0$ exists such that after t rounds of any execution of $\mathcal{A}$ starting in a Λ-faulty configuration, all subsequent configurations satisfy $\mathcal{S}_\mathcal{A}$.

The value of t is called the *containment time* of $\mathcal{A}$. The *fault-gap*[1] denotes the worst-case number of rounds any execution of $\mathcal{A}$ starting in a Λ-faulty configuration needs to reach a configuration satisfying $\mathcal{L}_\mathcal{A}$.

Definition 7.2 is slightly weaker than the definition provided in [DH95, DH97]. Dolev and Herman actually require the safety property to be satisfied immediately after the topology change. The motivation for this is that the model by Dolev and Herman permits so-called interrupt statements. The interrupt statement of an algorithm is executed immediately after any topology change and prior to any regular moves. They allow algorithms to keep track of topology changes without any delay. The mechanism that detects topology changes and executes the interrupt statements is assumed to be intrinsic to the distributed system. Such a mechanism may be an unrealistic assumption in the presence of state corruptions. We argue that state corruptions may affect the variables used for the detection of topology changes. Instead, we assume that the detection of topology changes is implemented as part of the super-stabilizing protocol itself and that super-stabilizing protocols run within the standard model without interrupt statements.

As an example of a super-stabilizing algorithm that uses interrupt statements, consider the vertex coloring algorithm proposed in [DH97]. Its safety property guarantees that no pair of neighboring nodes with the same color exists. A node may set its color to the special value $\perp$ which indicates that the node has not chosen any color yet. Clearly, if a link between two nodes of the same color recovers, the safety property would be violated. However, the interrupt statements of the algorithm set the color of one of the two nodes to $\perp$. In the standard model however, it would require a certain amount of time until the safety predicate can be satisfied. Furthermore, Dolev and Herman disregard the time needed to execute the interrupt statements in their model. Hence, we propose the definition as given in Definition 7.2. Whether it is possible to implement interrupt statements in the standard model is an open problem which is not considered in this thesis.

Super-stabilizing algorithms not relying on interrupt statements exist. One example is the super-stabilizing spanning tree algorithm given in [DH97]. Its safety predicate guarantees that the output is a valid span-

[1] corresponds to the super-stabilization time as defined in [DH95, DH97]

ning tree at all times. The algorithm ignores any recovering links and thus avoids reconstruction of the spanning tree. However, the class Λ of topology changes is assumed to exclude any failure of communication links that corresponds to edges within the computed tree. A technique to create super-stabilizing spanning tree algorithms is given by Blin et al. [BPBRT10]. The technique extends any existing spanning tree algorithm into one that has the safety property of maintaining loop-freedom during the reconstruction of the tree after a topology change. This is especially desirable if the tree is used for routing purposes. The class of topology changes for which the safety property is maintained is not restricted. The technique does not rely on interrupt statements.

Note that super-stabilization is very similar to the concept of safe convergence [KM06] (cf. Section 5.3.1). However, in the case of safe convergence, the safety property must be satisfied quickly starting from any initial configuration, regardless of whether it is the result of a state corruption or topology change. In the case of super-stabilization, the class of faults considered is often restricted, e.g., to a class of topology changes. Hence, super-stabilization should allow for stronger safety properties than safe convergence.

The focus of super-stabilization is on maintaining a safety property during convergence instead of reducing the time needed to restore correct output of the algorithm. That this is justified, is shown by the following examples. Cases are presented where almost every node $v \in V$ must adjust its local state after a topology change in order to reach a legitimate configuration. This holds even if the topological fault is a failure or recovery of a single link only. Hence quickly (i.e., within a number of rounds independent of n) converging to a correct output after a topological fault is in general not feasible.

Consider the graphs shown in Figure 7.1. The configurations shown are based on the algorithms presented in Section 5.1. However, the following discussion applies to any algorithm solving the corresponding problem. On the bottom of Figure 7.1, the communication link between nodes a and g fails. Node g was elected as the leader and is disconnected from the rest of the graph. Hence, all other $n - 1$ nodes have to select a new leader. Any leader election algorithm will thus have to adjust the local state of the remaining $n - 1$ nodes, before a legitimate configuration is reached. Furthermore, this can take up to $n - 2$ rounds.

In the topology in the middle of Figure 7.1, a communication link between nodes r and f recovers. A breadth-first spanning tree algorithm will adjust the parent nodes of at least nodes f, e, and d, since the new edge

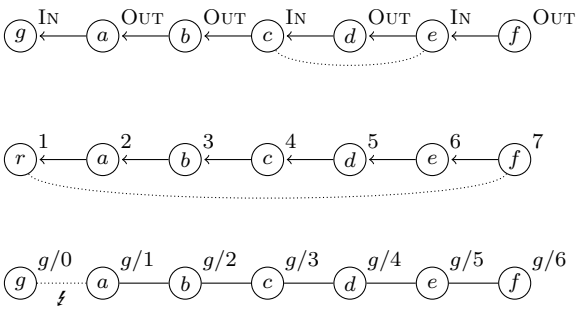


Figure 7.1: Three examples for topology changes: maximal independent set (top), breadth-first spanning tree (middle), and leader election (bottom)

is essentially a shortcut to the root node r. For a line graph with n nodes and a similar topology change, at least $\lfloor \frac{n-1}{2} \rfloor$ nodes have to change their output before the computed tree is a valid breadth-first tree. Again, this may take up to $\Omega(n)$ rounds.

As mentioned in Section 5.1, spanning tree construction and leader election are rather "global" problems. Next, we consider the maximal independent set problem. A link recovering does not necessarily render the set $\{v \mid v \in V \wedge v.s = \text{In}\}$ an invalid independent set. Hence, the case considered is where a link between two nodes in state In recovers. Assume the central scheduler and that all nodes execute protocol *MIS* as shown in Figure 2.1. Then, either node c or e changes its state to Out. In the case where node c changes state, node b will also change state from Out to In. It can be shown that in general, the recovery or failure of a single link does not require state changes outside the 2-hop neighborhood of the end nodes of the link.

7.2 Fault-Containing Super-Stabilization

In this section we define the notion of fault-containing super-stabilization. Informally speaking, it describes distributed algorithms that can cope with the scenario that a state corruption and a topology change occur simultaneously. However, if a state corruption or a topology change occur independently, a fault-containing super-stabilizing algorithm is to show the

same behavior as a fault-containing self-stabilizing or a super-stabilizing algorithm respectively. The challenge of fault-containing super-stabilization is to identify a topology change, even though the variables used to do so may have been corrupted.

Recall the definition of fault-containing self-stabilization as given in Section 5.2. Let $\mathcal{A}$ denote a distributed algorithm that is self-stabilizing with respect to a Boolean predicate $\mathcal{L}_\mathcal{A}$ and $\mathcal{L}_\mathcal{A}^{pri}$ a Boolean predicate over the primary variables of $\mathcal{A}$ such that

- $\mathcal{L}_\mathcal{A} \Rightarrow \mathcal{L}_\mathcal{A}^{pri}$ for all configurations $c \in \Sigma_\mathcal{A}$ and
- $\mathcal{L}_\mathcal{A}^{pri}$ is stable for any execution of $\mathcal{A}$ starting in a 1-faulty configuration.

If the predicate $\mathcal{L}_\mathcal{A}^{pri}$ is satisfied, then the output of $\mathcal{A}$ is considered correct. Recall that a self-stabilizing algorithm $\mathcal{A}$ is called *fault-containing* with respect to $\mathcal{L}_\mathcal{A}^{pri}$ if a constant $t \in \mathbb{N}_0$ exists such that $\mathcal{L}_\mathcal{A}^{pri}$ is satisfied after t rounds of any execution of $\mathcal{A}$ starting in a 1-faulty configuration. The value of t is called the *containment time* of $\mathcal{A}$. The *fault-gap* of $\mathcal{A}$ denotes the worst-case number of rounds any execution of $\mathcal{A}$ starting in a 1-faulty configuration needs to reach a configuration satisfying $\mathcal{L}_\mathcal{A}$.

Fault-containment focuses on restoring the correct output, i.e., $\mathcal{L}_\mathcal{A}^{pri}$ is satisfied, in constant time. As discussed in Section 7.1, a topology change makes it in general impossible to provide correct output within a constant number of rounds. Hence, the definition of fault-containing self-stabilization cannot be extended to the case where a topology change occurs. Thus, similar to plain super-stabilization, our definition of fault-containing super-stabilization focuses on reaching a safe configuration as soon as possible. In order to describe configurations that are the result of both a topological fault and a state corruption, we use the following notion:

Definition 7.3. An extended configuration $(E, c) \in \Sigma_\mathcal{A}^{ext}$ is called (Λ, k)-*faulty* if an extended configuration $(E', c') \in \Sigma_\mathcal{A}^{ext}$ exists such that (E', c') satisfies $\mathcal{L}_\mathcal{A}$ and

- E differs from E' in a single topology change of class Λ or
- c differs from c' in the local states of at most k nodes.

Note that k- and Λ-faulty configurations are also (Λ, k)-faulty. As in Section 7.1, let $\mathcal{S}_\mathcal{A}$ denote a Boolean predicate that reflects whether the a configuration satisfies the desired safety property. Furthermore, it is required that every legitimate configuration is also safe, i.e., $\mathcal{L}_\mathcal{A}$ implies $\mathcal{S}_\mathcal{A}$ for

all configurations of $\mathcal{A}$. Then fault-containing super-stabilization is defined as follows:

Definition 7.4 (Fault-Containing Super-Stabilization)**.** A distributed algorithm that is self-stabilizing with respect to $\mathcal{L}_\mathcal{A}$ is called *fault-containing* Λ-*super-stabilizing* with respect $\mathcal{S}_\mathcal{A}$ if

- $\mathcal{A}$ is fault-containing (as defined in Section 5.2) and
- a constant $t \in \mathbb{N}_0$ exists such that after t rounds of any execution of $\mathcal{A}$ starting in a $(\Lambda, 1)$-faulty configuration, all subsequent configurations satisfy $\mathcal{S}_\mathcal{A}$.

The value of t denotes the containment time of $\mathcal{A}$. The *fault-gap* of $\mathcal{A}$ denotes worst-case the number of rounds any execution of $\mathcal{A}$ starting in a $(\Lambda, 1)$-faulty configuration needs to reach a configuration satisfying $\mathcal{L}_\mathcal{A}$.

Note that in Section 5.2 and Definitions 7.2 and 7.4 the terms containment-time and fault-gap are defined in a consistent manner. In all three contexts, the containment-time indicates how well the algorithm at hand can cope with a fault, i.e., how fast the desired property (correct output or safety) holds, and the fault-gap denotes the minimal time between two faults that can be handled in the desired way. This is visualized in Figure 7.2. Also note that Definition 7.4 implies that $\mathcal{A}$ is super-stabilizing, as the set of all $(\Lambda, 1)$-faulty configurations also includes all Λ-faulty configurations.

7.3 The Transformation

This chapter presents a transformation that will convert an algorithm satisfying Definition 7.2 to an algorithm that satisfies Definition 7.4. The transformation increases the containment-time by a constant number of rounds only. The input of the transformation consists of a distributed algorithm $\mathcal{A}$ that is self-stabilizing with respect to $\mathcal{L}_\mathcal{A}$ and Λ-super-stabilizing with respect to $\mathcal{S}_\mathcal{A}$, where Λ denotes the class of topology changes that either add or remove a single edge and $\mathcal{S}_\mathcal{A}$ denotes the safety predicate of $\mathcal{A}$. All variables of $\mathcal{A}$ are defined to be primary. All variables added by the transformation are secondary. The output of the transformation is the distributed algorithm $\mathcal{A}_{FS}$, which will be shown to be silent and fault-containing super-stabilizing with respect to $\mathcal{S}_\mathcal{A}$.

$\mathcal{A}_{FS}$ restores the values of corrupted primary variables prior to the corruption. At the same time, it prevents nodes within the neighborhood of the node affected by the state corruption from executing moves of $\mathcal{A}$. After

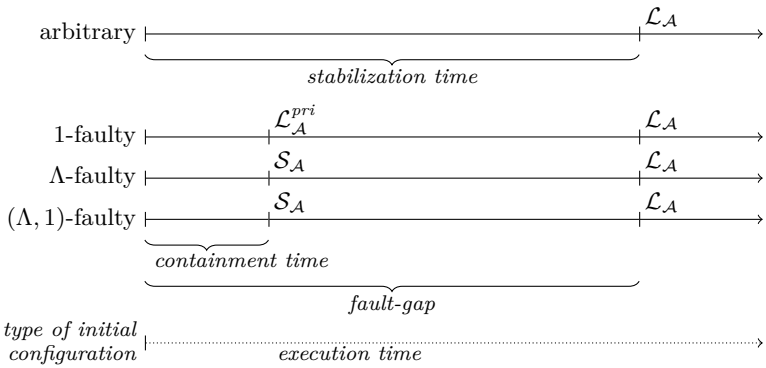


Figure 7.2: Visualization of the different refinements of self-stabilization. From top to bottom: self-stabilization, fault-containing self-stabilization, super-stabilization, fault-containing super-stabilization

the corruptions are repaired, moves of $\mathcal{A}$ are executed. If that is successful, then the transformed algorithm inherits the super-stabilization property of $\mathcal{A}$. We assume that $\mathcal{A}$ implements the detection of topology changes without interrupt statements. Hence, the primary variables must include all information required by $\mathcal{A}$ to detect the topology change and subsequently behave in a super-stabilizing way. However, it is sometimes impossible to restore the original values of the corrupted primary variables. At least two backups on distinct nodes are required to decide whether a single state corruption has affected the primary variables of a node $v \in V$ or one of the backups. Problems arise from connected components of only two nodes or isolated nodes. Such components will be called *minor components*. If minor components exist prior to or after the topology change, then two backups cannot be provided. A $(\Lambda, 1)$-faulty configuration may thus be ambiguous in the sense that the original values of the primary variables are not uniquely determined. This poses one of the main challenges for the transformation. This is discussed in depth in Section 7.3.7.

Figure 7.3 shows the layered architecture of our transformation described in this section. The upper layer implements the detection and the repair of state corruptions. If no corruptions are detected or all have been repaired, then the upper layer executes moves of $\mathcal{A}$, the algorithm to be transformed.

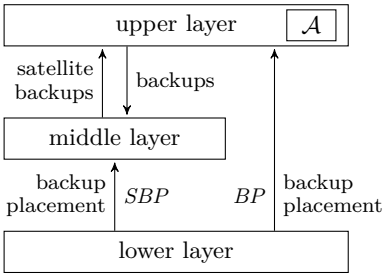
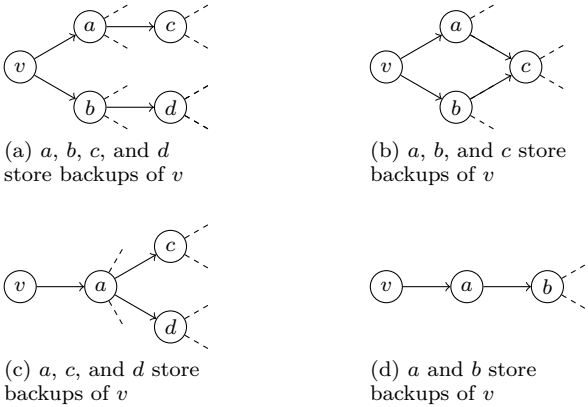


Figure 7.3: General architecture

Per node, the upper layer keeps backups of the variables of algorithm $\mathcal{A}$ on up to three direct neighbors. The backups allow for detecting and repairing faults in most cases. The main challenges arise from nodes with less than three neighbors and backups becoming unavailable due to topology changes. For these cases, the middle layer provides up to 2 additional backups within distance 2 of each node. The upper and middle layer are implemented using the technique for local synchronization based on cells as presented in Section 5.8. However, the implementation of cells is revised to deal with topology changes more gracefully.

The lower layer computes the nodes on which the backups are to be placed. It is assumed to be a regular silent self-stabilizing algorithm. However, the lower layer is not in the focus of this chapter. It is seen as a black box which provides its output in the form of two functions *BP* and *SBP*. Section 7.3.1 describes the requirements for the placement of the backups. It ensures that at least 2 backups are available after the removal of a single edge, provided that the topology allows for it. Section 7.3.3 describes the refined implementation of cells. The interface with the lower layer is described in Section 7.3.4. To prove the feasibility of an implementation of a lower layer that satisfies the placement requirements as outlined in Section 7.3.1, we give an algorithm for computing a random placement. Furthermore, it is outlined how the algorithm described in Chapter 6 can be modified to compute a placement with minimal variance suitable for this transformation. Sections 7.3.5 and 7.3.6 describe the implementation of the upper and middle layer. The repair of corruptions by the upper layer is described in more detail in Section 7.3.7.



(a) a, b, c, and d store backups of v

(b) a, b, and c store backups of v

(c) a, c, and d store backups of v

(d) a and b store backups of v

Figure 7.4: Backup placement for the case $\deg(v) < 3$

7.3.1 Backup Placement Requirements

To restore the values of a corrupted primary variable it is vital to have access to at least two backups on different nodes. In particular the failure of a communication link can cause backups to become unreachable. This sections describes the strategy for placing four backups within the 2-hop neighborhood in such a way that two backups remain reachable even if one communication link fails.

First consider a node $v \in V$ with $\deg(v) \geq 3$. Placing three backups of the primary variables of node v on three different neighbors is sufficient. After the failure of a single link, two backups remain accessible. A different strategy to satisfy this requirement is needed for nodes with a degree of less than 2.

Clearly, if a node $v \in V$ is part of a minor component after the topology change, then it is impossible to provide two backups accessible by v on two different nodes. The same holds if node v is part of a minor component prior to the topology change. Hence, this case is excluded from the following discussion.

Consider the case of a node $v \in V$ with $\deg(v) = 2$. Example topologies are depicted in Figures 7.4a and 7.4b. To satisfy the requirement that at least two backups remain accessible, one has to distribute up to four backups within the 2-hop neighborhood of v. In Figure 7.4a, backups are

kept on nodes a, b, and on one neighbor of each a and b. Note that the removal of a single edge (e.g., edge (v, a)) may cause two backups to become inaccessible. Hence, four is the minimal number of backups. We ask the reader to verify that after removing an arbitrary single edge, at least 2 backups are reachable within the 2-hop distance of node v, unless node v is part of a minor component after the removal of the edge. Figure 7.4b illustrates that a common neighbor of a and b can be chosen to store a third backup. In this case, no fourth backup is needed.

Figures 7.4c and 7.4d illustrate the case $\deg(v) = 1$. If possible, up to two different neighbors of a (excluding v) store a backup in addition to node a. Again, two backups remain accessible by node v, unless v is part of a minor component after the removal of a single edge.

Backups of the primary variables of node v at distance 2 of node v are called *satellite backups*, while the backups at distance 1 of v are called *adjacent backups*. The upper layer is responsible for managing all adjacent backups, while the middle layer manages all satellite backups.

7.3.2 Spawning and Deleting Instances

Before the revised cell mechanism is presented in Section 7.3.3, we define the necessary operations to dynamically spawn and delete instances. For the implementation of cells as described in Section 5.8, it was simply assumed that all responding instances (u, R_v) with $u \in V$ and $v \in N(v)$ exist at all times. It was not discussed, what happens after a topology change, in particular the recovery of a link. If a link $(v, u) \in E$ recovers, node u must reserve space for the local state of a new responding instance (u, R_v) in its memory. Since $\mathcal{A}_{FL}$ presented Section 5.8 is self-stabilizing, the local state of a responding instance could be initialized with arbitrary values.

For the transformation described in this chapter, an initialization with arbitrary values would not be suitable. Furthermore, the implementation explicitly decides at which points responding instances may be spawned to avoid resets of the local phase clock provided by cells.

Each node executes a fixed and a dynamic set of protocols. The two sets are assumed to be disjoint. The fixed set of protocols is defined by the distributed algorithm, i.e., for a given algorithm $\mathcal{A}$ this fixed set of node v is equal to $\mathcal{A}(v)$. Protocols may *spawn* and *delete* instances at runtime. For this purpose we define the operations *spawn*(P) and *delete*(P). If executed by an instance on node v, the operations spawn or delete the instance (v, P). The dynamic set of protocols executed by node v is denoted by $K(v)$. When (v, P) is spawned, then P is added to $K(v)$ and removed from $K(v)$ when

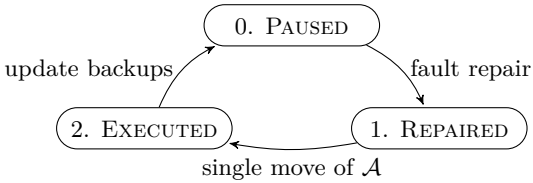


Figure 7.5: States and transitions of cells

(v, P) is deleted, or in other words: $P \in K(v)$ if and only if (v, P) has been spawned and was not deleted yet. It is said that an instance (v, P) *exists* if and only if either $P \in K(v) \cup \mathcal{A}(v)$. If an instance has been spawned, it can be selected by the scheduler in any subsequent step. If an instance is deleted, the change becomes effective at the end of the step, i.e., the instance may make a move and be deleted in the same step.

During the spawning of an instance, its variables are initialized according to the information provided along with the protocol implementation. If a node's state is corrupted, this may also affect the dynamic set: The corruption can delete any number of instances or spawn additional instances with an arbitrary local state. The fixed set $\mathcal{A}(v)$ is assumed to be incorruptible. In addition to that, a state corruption may perturb all variables of any instance (fixed or dynamic).

7.3.3 Refined Cells

This section presents a refined implementation of the cells presented in Section 5.8. As in Section 5.8, cells are used as a form of local synchronization. A cell C_v, $v \in V$ consists of the *center instance* (v, Q) and all existing *responding instances* (u, R_v), $u \in N(v)$. Note that protocol R_v is parametrized. The parameter v determines the center instance (v, Q) that (u, R_v) interacts with. Two cells C_v and C_u are called *neighbors* if and only if u and v are neighbors. Figure 7.6 shows the implementation of protocols Q and R_v. They implement a *dialog* between the center and all responding instances.

Each cell is a locally distributed implementation of the state-machine shown in Figure 7.5. The state-machine is identical to the one used in Section 5.8. Again, the three states of the state-machine are called *positions* in order to avoid confusion with the notion of local states. We assign an integer value as shown in Figure 7.5 to each position. Whenever it is con-

Protocol Q

Nodes: v is the current node
Variables: $v.s, v.q \in \{\textsc{Paused}, \textsc{Repaired}, \textsc{Executed}\}$

do

[Q1] $\neg \mathit{dialogConsistent}(v) \wedge (v.s, v.q) \neq (\textsc{Paused}, \textsc{Paused}) \longrightarrow$
$(v.s, v.q) := (\textsc{Paused}, \textsc{Paused})$

[Q2] $\square\ \mathit{dialogPaused}(v) \wedge (\mathit{startUpper}_Q(v) \vee \mathit{startMiddle}_Q(v)) \longrightarrow$
$v.q := \textsc{Repaired}$

[Q3] $\square\ \mathit{dialogAcknowledged}(v) \longrightarrow$
$\mathit{actionMiddle}_Q(v);\ \mathit{actionUpper}_Q(v);\ v.s := v.q;$
if $v.s \neq \textsc{Paused} \vee \mathit{startUpper}_Q(v) \vee$
$\mathit{startMiddle}_Q(v)$ **then** $v.q := (v.s + 1) \bmod 3$

od

Protocol R_v

Nodes: u is the current node, v is the center node
Variables: $u.r_v \in \{\textsc{Paused}, \textsc{Repaired}, \textsc{Executed}\}$
On Spawn: $u.r_v := \textsc{Paused}$

do

[R1] $v.s = v.q = \textsc{Paused} \wedge u.r_v \neq \textsc{Paused} \longrightarrow u.r_v := \textsc{Paused}$

[R2] $\square\ \mathit{validQuery}(v) \wedge u.r_v = v.s \wedge (v.q = \textsc{Executed} \Rightarrow \neg \mathit{repaired}(u))$
$\longrightarrow \mathit{actionMiddle}_{R_v}(u);\ \mathit{actionUpper}_{R_v}(u);\ u.r_v := v.q$

od

$$
\begin{aligned}
R(v) &:= \{u \in N(v) \mid R_v \in K(u)\} \\
\mathit{operational}(v) &\equiv N(v) \neq \emptyset \Rightarrow R(v) \neq \emptyset \\
\mathit{validQuery}(v) &\equiv v.q = (v.s + 1) \bmod 3 \\
\mathit{dialogConsistent}(v) &\equiv (v.s = v.q = \textsc{Paused} \vee \mathit{validQuery}(v)) \wedge \\
&\quad \mathit{operational}(v) \wedge (\forall u \in R(v) : u.r_v \in \{v.s, v.q\}) \\
\mathit{dialogPaused}(v) &\equiv v.s = v.q = \textsc{Paused} \wedge \mathit{operational}(v) \wedge \\
&\quad (\forall u \in R(v) : u.r_v = \textsc{Paused}) \\
\mathit{dialogAcknowledged}(v) &\equiv \mathit{validQuery}(v) \wedge \mathit{operational}(v) \wedge \\
&\quad (\forall u \in R(v) : u.r_v = v.q) \wedge (v.s = \textsc{Paused} \Rightarrow R(v) = N(v))
\end{aligned}
$$

Figure 7.6: Cell implementation

Protocol M

Nodes: v is the current node

do

$(\exists R_u \in K(v) : u \notin N(v)) \vee$
$(\exists u \in N(v) : R_u \notin K(v) \wedge u.s = \text{PAUSED}) \longrightarrow$
foreach $R_u \in K(v) : u \notin N(v)$ **do** *delete*(R_u);
foreach $u \in N(v) : R_u \notin K(v) \wedge u.s = \text{PAUSED}$ **do**
spawn(R_u)

od

Figure 7.7: Rule for spawning and deleting responding-instances

venient, we will use the integer value interchangeably with the name of the position, e.g., 0 corresponds to PAUSED. The current position of the cell is stored in the variable $v.s$. In order to advance the current position, (v, Q) must get the permission of all responding instances. For this purpose, (v, Q) first sets the variable $v.q$ to the desired position. The pair $(v.s, v.q)$ is said to be a *query* for a transition from $v.s$ to $v.q$ if $v.q \neq v.s$. Otherwise, the pair is said to be a *pause* at position $v.s$. Each responding instance (u, R_v) maintains a *response-variable* $u.r_v$. It is used to acknowledge queries made by the center instance. An acknowledgment is given by assigning the value of $v.q$ to $u.r_v$. If the query is acknowledged by all responding instances, then (v, Q) sets $v.s$ to the value of $v.q$ and thereby completes the transition to the desired state. Note that queries for a transition not shown in Figure 7.5 and pauses at a position other than PAUSED are regarded as invalid. Hence, (v, Q) also sets $v.q$ if $v.s \neq \text{PAUSED}$, so that $(v.s, v.q)$ forms a valid query or pause at all times. A *cycle* of a cell starts with the transition from PAUSED to REPAIRED and ends with the transition from EXECUTED to PAUSED.

One major difference to the implementation given in Section 5.8 is that within a cell C_v some responding (u, R_v) may not exist. The set $R(v)$ denotes the set of neighbors $u \in N(v)$ which currently execute protocol R_v. The missing responding instances are spawned by protocol M as shown in Figure 7.7 between cycles, i.e., only if $v.s = \text{PAUSED}$. Furthermore, an instance (u, M) makes sure that all instances (u, R_v) with $v \notin N(u)$ are deleted eventually.

The predicate *dialogConsistent*(v) (see Figure 7.6) describes the absence of faults in the dialog of cell C_v. It is said that C_v is *dialog-consistent*

if $(v.s, v.q)$ forms a valid pause or query and all responding instances either acknowledge the previous query for a transition to the current position of C_v ($u.r_v = v.s$) or acknowledge the current query ($u.r_v = v.q$). Furthermore, it is required that the cell is *operational*, i.e., at least one responding instance exists unless $N(v) = \emptyset$. Lemma 7.10 shows that the predicate *dialogConsistent*(v) is stable, i.e., moves of protocols Q, R_v, or M never render a previously dialog-consistent cell non-dialog-consistent. Note that when a responding-instance is spawned, the initialization of the response-variables with PAUSED ensures that C_v remains dialog-consistent. The predicate *dialogAcknowledged*(v) describes a *dialog-acknowledged* cell, which is a dialog-consistent cell where $(v.s, v.q)$ is a valid query and all responding-instances acknowledge the query. Furthermore, for a cell to be dialog-acknowledged, it is required that $R(v) = N(v)$ if $v.s = \text{PAUSED}$. This ensures that a cell (v, Q) waits at position $v.s = \text{PAUSED}$ until all responding-instances have been spawned by protocol M. A *dialog-paused* cell denotes a dialog-consistent cell with $v.s = v.q = \text{PAUSED}$ (cf. predicate *dialogPaused*(v)). This implies that all response-variables also have the value PAUSED. Non-operational cells occur in particular if a topology change adds an edge that connects a single node to an existing graph component or if the only instance of R_v within C_v is deleted by a state corruption.

It will be shown that the given dialog-implementation is resilient to state corruptions and topology changes in the following sense: For any execution starting in a $(\Lambda, 1)$-faulty extended configuration (E, c), it holds for any cell C_v, $v \in V$ with $N(v) \neq \emptyset$ in E that

- PAUSED to REPAIRED is the first transition that C_v executes, with the exception of (v, Q) executing the move that completes the transition EXECUTED $\to$ PAUSED beforehand, and
- prior to completing the transition PAUSED $\to$ REPAIRED, all missing responding instances are spawned, i.e., $R(v) = N(v)$, and all of them execute a move to acknowledge the transition.

Note that only such configurations are considered legitimate in which all cells are dialog-paused. Thus, for operational cells, the corruption of a single node's state can easily be detected. If a responding instance is affected by the state corruption, then it holds that $v.s = v.q = \text{PAUSED}$ and a corrupted response variable $r_v \neq \text{PAUSED}$ causes a reset to $r_v := \text{PAUSED}$ via Rule R1. If $v.s$ or $v.q$ is corrupted, anything other than $(v.s, v.q) \in \{(\text{EXECUTED}, \text{PAUSED}), (\text{PAUSED}, \text{REPAIRED})\}$ is detected, and is reset to

$v.s = v.q =$ PAUSED via Rule Q1 as it violates dialog-consistency of C_v. Since non-operational cells are also considered as non-dialog-consistent, they perform a reset to $v.s = v.q =$ PAUSED as well.

The implementation of the upper and middle layer is based on the revised cell implementation. Both layers contribute the predicates $\mathit{startUpper}_Q(v)$ and $\mathit{startMiddle}_Q(v)$ and the procedures $\mathit{actionUpper}_Q$, $\mathit{actionUpper}_{R_v}$, $\mathit{actionMiddle}_Q$, and $\mathit{actionMiddle}_{R_v}$. They are explained in detail in Sections 7.3.5 and 7.3.6. $\mathit{startUpper}_Q(v)$ and $\mathit{startMiddle}_Q(v)$ are Boolean predicates that control whether a dialog-paused cell C_v starts a new cycle (Rule Q2). Furthermore, the two conditions have also been incorporated into Rule Q3. To avoid an unnecessary execution of Rule Q2, Rule Q3 starts a new cycle straight away if any of the two configurations is satisfied. The procedures $\mathit{actionUpper}_Q$ and $\mathit{actionUpper}_{R_v}$ as well as $\mathit{actionMiddle}_Q$ and $\mathit{actionMiddle}_{R_v}$ implement the transition actions that are executed during each transition. They are invoked by any instance of R_v and Q prior to the acknowledgments and prior to actually changing the position of the cell. This mechanism is used by the upper and middle layer to either prepare data during a move of the responding instance that is then used by the center instance or vice versa. This allows the center and responding instances of cell C_v to indirectly access information from v's 2-hop neighborhood.

The predicate $\mathit{repaired}(u)$ controls whether acknowledgments during the transition from REPAIRED to EXECUTED are delayed. The predicate is defined as part of the description of the upper layer in Section 7.3.5. As in Section 5.8, delaying acknowledgments is used to prevent premature execution of moves of algorithm $\mathcal{A}$, the input to the transformation. Informally, the predicate $\mathit{repaired}(u)$ indicates whether the primary variables of node u are safe to use during the execution of $\mathcal{A}$ by any neighbor of u.

A cell C_v is called *blocked* if it is dialog-consistent, $v.q =$ EXECUTED, at least one responding instance (u, R_v) exists such that $u.r_v = v.s$, and $\neg\mathit{repaired}(u)$ is satisfied. Clearly, cell C_v cannot finish the transition to EXECUTED unless the predicate $\mathit{repaired}(u)$ becomes true. This bears the potential for deadlocks. However, it will be shown in Lemma 7.16 that *repaired* becomes true for all cells eventually.

Note that the state-machine and the rules of Q and R_v are mostly identical to those which can be found in Section 5.8. The refinement consists of the modifications to the predicate $\mathit{dialogConsistent}(v)$ and the derived predicates $\mathit{dialogPaused}(v)$ and $\mathit{dialogAcknowledged}(v)$, which incorporate the new notion of an operational cell and the fact that responding instances are spawned at runtime.

7.3.4 Lower Layer

This section defines the interface between the lower layer and the other two layers. The purpose of the lower layer is to compute a placement of the adjacent and satellite backups of each node that satisfies the requirements given in Section 7.3.1. The lower layer is assumed to be self-stabilizing and silent. In order to show that this is feasible, an algorithm to compute an arbitrary placement is given. Furthermore, it is discussed how the placement algorithm given in Chapter 6 could be extended into an implementation of the lower layer.

The interface between the lower layer and the other two layers consists of the two deterministic functions $BP(v)$ (backup distribution) and $SBP(v)$ (satellite backup distribution) that only read variables of the lower layer. The return values of both functions are required to be locally computable by node v, i.e., the return value can be computed solely based on variables within $N[v]$.

The function $BP(v)$ is used by the upper layer. It returns the set of neighbors of v that are supposed to store adjacent backups of v's primary variables. It is assumed to satisfy $BP(v) \subseteq N(v)$ and $|BP(v)| \leq 3$ at all times. Eventually, i.e., when the lower layer has terminated, it is required to return exactly three neighbors of v or $BP(v) = N(v)$ if v has less than three neighbors. In addition, the upper layer requires that $BP(v) \cap R(v) \neq \emptyset$ is satisfied at all times for all $v \in V$ with $R(v) \neq \emptyset$. If necessary, the function may (deterministically) choose an arbitrary neighbor $u \in R(v)$. The requirement is discussed in more detail in Section 7.3.5.

The function $SBP(v)$ is used by the middle layer. It returns a set of pairs (w, u) with $w, u \in N(v)$ and $w \neq u$. A pair $(w, u) \in SBP(v)$ indicates that node u is to store a satellite backup of the primary variables of w. Eventually, $SBP(v)$ must satisfy that

$$(\#(a, b) \in SBP(v) : a = w) = \begin{cases} 2 & \text{if } \deg(w) = 1 \\ 1 & \text{if } \deg(w) = 2 \\ 0 & \text{otherwise} \end{cases}$$

for all $v \in V$ and $w \in N(v)$. During stabilization of the lower layer, $SBP(v)$ is allowed to return a different number of pairs (a, b) with $a = w$, but never more than 2 for each $w \in N(v)$. If not enough neighbors of node v exist to satisfy the above requirements $SBP(v)$ must eventually return as many pairs as possible without exceeding the above requirements. Such a case is depicted in Figure 7.4d. Since node a has only two neighbors, $SBP(a)$ cannot contain two distinct pairs (v, u) with $u \in N(v)$ and $u \neq v$.

Protocol K

Variables: $v.k$: Integer in the range $[0, 3]$

do $v.k \neq k(v) \longrightarrow v.k := k(v)$ **od**

Protocol P

Variables: $v.\beta$: set of at most 3 node identifiers
$v.\gamma$: set of at most 2Δ pairs (w, u) with $w \neq u$, where w and u are node identifiers

do
$v.\beta \not\subseteq N(v) \vee |v.\beta| \neq k(v) \vee$
$(\exists (a, b) \in v.\gamma : a \notin N(v) \vee b \notin N(v)) \vee$
$(\exists w \in N(v) : s(v, w) \neq (\#(a, b) \in v.\gamma : a = w)) \longrightarrow$
choose $P_v \subseteq N(v)$ with $|P_w| = k(v)$;
foreach $w \in N(v)$ **do**
choose $P_w \subseteq N(v) \setminus \{w\}$ with $|P_w| = s(v, w)$;
end
$v.\beta := P_v$; $v.\gamma := \bigcup_{w \in N(w)} \{w\} \times P_w$
od

$$k(v) := \min(3, \deg(v))$$

$$s(v, u) := \begin{cases} \min(2, \deg(v) - 1) & \text{if } u.k = 1 \\ \min(1, \deg(v) - 1) & \text{if } u.k = 2 \\ 0 & \text{otherwise} \end{cases}$$

Figure 7.8: Protocols for computing a random backup placement

We proceed to describe the silent self-stabilizing distributed algorithm $\mathcal{A}_{PR}$ which computes a random backup placement that satisfies the above requirements. It holds $\mathcal{A}_{PR}(v) = \{K, P\}$ for all nodes $v \in V$, where protocol K and P are as shown in Figure 7.8.

An instance (v, K) updates the variable $v.k$ of each node, and sets it to the value of $k(v)$. The variable allows protocol P to gain information about the degree of the neighbors of each node. This is necessary to know how many satellite backups are required. Each (v, P) updates $v.\beta$ and $v.\gamma$ whenever they don't match the requirements as given above. The function

$s(v, u)$ returns the number of satellite backups of the primary variables of node u that node v should place among its neighbors.

For $\mathcal{A}_{PR}$, $SBP(v)$ is defined to return $v.\gamma$. The value returned by $BP(v)$ is equal to $v.\beta$, unless $v.\beta \cap R(v) = \emptyset$ and $R(v) \neq \emptyset$. Then, $BP(v)$ deterministically selects an arbitrary node $u \in R(v)$, e.g., the one with the largest identifier. To show that $\mathcal{A}_{PR}$ is silent and self-stabilizing is simple. The algorithm may be seen as a collateral composition. Each instance (v, K) makes at most one move. Between two moves of any instance of K, any instance (v, P) can make at most one move.

A local 3-placement as computed by algorithm $\mathcal{A}_P$ described in Chapter 6 can be used to compute a placement of the adjacent backups with local minimum variance. However, the algorithm does not consider the placement of the satellite backups. In order to make the algorithm compute a placement of the satellite backups, it is necessary to add protocol K to $\mathcal{A}_P$. Then protocol P_Q as shown in Figure 6.5 can be extended to place $s(v, u)$-many satellite backups in the neighborhood of v. The satellite backups should be moved from one neighbor of v to another if the load of the two differs by at least 2. For moving the satellite backups, the existing query/answer mechanism can be used.

However, the placement of the satellite backups is restricted. This restriction can lead to a scenario as shown in Figure 7.9 on the left. The arrows denote the placement of the 3 backups of the primary variables of v. The arrow next to edge (v, e) denotes the placement of the satellite backup of the primary variables of node a (i.e., $(a, e) \in SBP(v)$). The neighbors of nodes a to e are omitted in the figure. The numbers next to the nodes denote their load. It would be desirable to move the satellite backup from node e to node a, as their load differs by at least 2. This would lower the variance of the loads, as discussed in Chapter 6. However, the satellite backups of node a's primary variables must not be placed on node a itself.

A strategy to lower the variance in such a situation is to first move the satellite backup to a neighbor that stores an adjacent backup and has a lower load. For the example shown in Figure 7.9, the satellite backup is moved from node e to node c. This does not change the variance of the load, as the loads of nodes c and e differ by 1 only. Then, the adjacent backup stored by node c is moved to node a. The result is shown in Figure 7.9 on the right. We are not aware of a counter example, for which this strategy does not achieve (local) minimum variance.

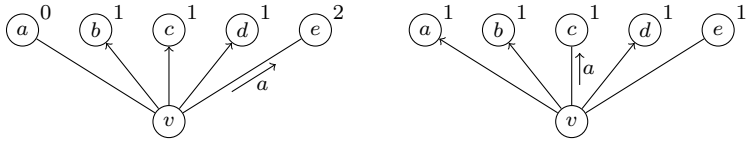


Figure 7.9: Impact of constraints on satellite backup placement

7.3.5 Upper Layer

This section describes the implementation of the upper layer. This layer manages the adjacent backups of the primary variables of a node $v \in V$. They are moved to the neighbors selected by the lower layer. Furthermore, the upper layer is responsible for repairing corruptions of primary variables as well as executing moves of algorithm $\mathcal{A}$, the input to the transformation. Without loss of generality, we assume that algorithm $\mathcal{A}$ uses only a single variable $v.p \in \sigma_{\mathcal{A}}$ per node $v \in V$. Furthermore, we use $G_{\mathcal{A}}$ and $S_{\mathcal{A}}$ as defined in Section 2.7.

The implementation of the upper layer consists of the two procedures $actionUpper_Q$ and $actionUpper_{R_v}$ which are called by the cell mechanism as described in Section 7.3.3. They require additional variables. The implementation, including a list of the additional variables and their domain, is shown in Figure 7.10. The variables are added to the local states of (v, Q) and each (u, R_v), $u \in N(v)$ respectively. Variable $v.p$ is the only primary variable of v and stores the local state of node v with respect to algorithm $\mathcal{A}$. Variable $v.fp$ holds a pointer to a neighbor of v in form of the neighbor's identifier. Its use is explained in Section 7.3.7.3. The set $v.bptr$ holds a set of at most 3 such pointers. A responding instance (u, R_v) is supposed to store an adjacent backup of $v.p$ if and only if $u \in v.bptr$. The backup is stored in the variable $u.b_v$. This variable can also assume the value $\perp$, which indicates that no backup is held by (u, R_v). The *decision-variable* $u.d_v$ is used by $actionUpper_{R_v}$ to provide information that $actionUpper_Q$ needs during the repair of $v.p$. Note that when an instance (u, R_v) is spawned, then $u.b_v$ is initialized with value $\perp$. Furthermore, it is assumed that the value $\perp$ occupies only $\mathcal{O}(1)$ bits in the local state of (u, R_v). Note that for notational convenience, any reference to a backup of a non-existing responding-instance, i.e., $u.b_v$ with $R_v \notin K(u)$, is assumed to yield $\perp$.

Procedure $actionUpper_Q$ implements the detection and repair of a corrupted primary variable $v.p$ during the transition Paused $\rightarrow$ Repaired.

If $v.q = \text{REPAIRED}$, $actionUpper_{R_v}$ sets the variable $u.d_v$ which contains information about the satellite backups or the topology. During the transition $\text{REPAIRED} \rightarrow \text{EXECUTED}$, $actionUpper_Q$ executes individual moves of $\mathcal{A}$ for node v and updates the variable $v.bptr$ with the backup distribution computed by the lower layer. During the transition to PAUSED, each responding instance (u, R_v) invokes $actionUpper_{R_v}$ which sets the adjacent backup $v.b_v$ to the value of $v.p$ or to $\perp$ depending on whether $u \in v.bptr$.

We define a predicate *repaired*(v) as given in Figure 7.11. It describes the absence of corruptions of the dialog variables, the primary variable $v.p$, and the adjacent backups within C_v. If *repaired*(v) is satisfied then C_v is said to be *repaired*. Dialog-consistency of C_v as defined in Section 7.3.3 reflects the absence of faults in the dialog of C_v. One indicator for the absence of corruptions in the primary variable and backups is if $v.p$ and all existing backups have the same value and at least one backup exists unless $N(v) = \emptyset$. If that is the case, then C_v is called *backup-consistent*. However, backup-consistency is not a stable property. In fact, a cell loses this property regularly, e.g., after a move of $\mathcal{A}$ altered $v.p$ and the backups have not been updated yet. Furthermore, if a cell has completed the transition to position REPAIRED, this means that it has completed the repair of $v.p$. Hence, *repaired*(v) is also satisfied if C_v is dialog-consistent and its current position is REPAIRED or EXECUTED. However, in the latter case the backups held by the responding instances that have already acknowledged the query for a transition to PAUSED may be corrupted. Therefore, *repaired*(v) requires that all responding instances that have already acknowledged hold no backup or a backup equal to $v.p$. Furthermore, unless $N(v) = \emptyset$, there must exist at least one responding instance (u, R_v) which satisfies either

- $u \in v.bptr$ and (u, R_v) has not acknowledged the transition to PAUSED yet or
- (u, R_v) has acknowledged the transition already and $u.b_v = v.p$.

In the first case, the next move of (u, R_v) sets $u.b_v := v.p$. Note that if *repaired*(v) is satisfied, cell C_v is guaranteed to be backup-consistent when it reaches position PAUSED. The requirement $BP(v) \cap R(v) \neq \emptyset$ guarantees that $v.bptr \cap R(v) \neq \emptyset$ if the transition to EXECUTED is completed and thus that at least one backup of $v.p$ is created during the transition to PAUSED. This is necessary to ensure that *repaired*(v) is a stable predicate.

Furthermore, we define the predicate $startUpper_Q(v)$, which decides when a cell leaves the dialog-pausing state and starts a new cycle. The predicate is satisfied if the current distribution of backups does not match the value

Procedure $actionUpper_Q(v)$

Nodes: v is the current node
Variables: $v.p \in \sigma_{\mathcal{A}}$; $v.fp$: node identifier ID or $\perp$;
$v.bptr$: set of at most three node identifiers

1 **if** $v.q = $ Repaired **then**
2 $v.fp := formerPeer(v)$
3 **if** $\deg(v) = 1 \wedge \exists u \in R(v) : u.d_v = $ Single **then** $pairReset(v, u)$
4 **else if** $\exists u \in R(v) : u.b_v = v.p \vee u.d_v = $ Keep **then** Keep value of $v.p$
5 **else if** $\exists u, w \in R(v) : u \neq w \wedge u.b_v = w.b_v \wedge u.b_v \neq \perp$ **then** $v.p := u.b_v$
6 **else if** $\exists u \in R(v) : u.b_v \neq \perp \wedge u.d_v = $ Update **then** $v.p := u.b_v$
7 **else if** $\deg(v) \geq 2 \wedge \exists u \in R(v) : u.id = v.fp \wedge u.d_v \in \{$Single, Minor$\}$ **then** $pairReset(v, u)$
8 **else if** $\deg(v) = 1 \wedge \exists u \in R(v) : u.d_v = $ Minor **then** $pairReset(v, u)$
9 **else if** $singleEnabled_{\mathcal{A}}(v : v.p)$ **then**
10 Compute new value of $v.p$ such that $\neg singleEnabled_{\mathcal{A}}(v : v.p)$
11 **end**
12 **end**
13 **if** $v.q = $ Executed **then**
14 $v.bptr := BP(v)$
15 **if** $\deg(v) = 1 \wedge \exists u \in R(v) : u.d_v = $ Single **then** Keep value of $v.p$
16 **else if** $G_{\mathcal{A}}$ **then** $S_{\mathcal{A}}$
17 **end**

Procedure $actionUpper_{R_v}(u)$

Nodes: u is the current node, v is the center node
Variables: $u.b_v \in \sigma_{\mathcal{A}} \cup \{\perp\}$;
$u.d_v \in \{$None, Keep, Update, Minor, Single$\}$
On Spawn: $u.b_v := \perp$; $u.d_v :=$ None

1 **if** $v.q = $ Repaired **then**
2 **if** $\deg(u) = 1$ **then** $u.d_v :=$ Single
3 **else if** $\exists w \in N(u) : w \neq v \wedge w.sb_{u,v} = v.p$ **then** $u.d_v :=$ Keep
4 **else if** $u.b_v \neq \perp \wedge \exists w \in N(u) : w \neq v \wedge w.sb_{u,v} = u.b_v$ **then** $u.d_v :=$ Update
5 **else if** $actualBP(u) = \emptyset \wedge repaired(u) \wedge v.id = u.fp$ **then** $u.d_v :=$ Minor
6 **else if** $(actualBP(u) \neq \emptyset \vee \neg repaired(u)) \wedge v.id = formerPeer(u)$ **then** $u.d_v :=$ Minor
7 **else** $u.d_v :=$ None
8 **end**
9 **if** $v.q = $ Paused **then**
10 **if** $u \in v.bptr$ **then** $u.b_v := v.p$ **else** $u.b_v := \perp$
11 **end**

Figure 7.10: Implementation of the upper layer

$$
\begin{aligned}
\mathit{backupConsistent}(v) \equiv\ & (\forall u \in R(v) : u.b_v \in \{v.p, \bot\}) \ \wedge \\
& (N(v) \neq \emptyset \Rightarrow \exists u \in R(v) : u.b_v = v.p) \\
\mathit{repaired}(v) \equiv\ & \mathit{dialogConsistent}(v) \ \wedge \ (\mathit{backupConsistent}(v) \ \vee \\
& v.s = \textsc{Repaired} \ \vee \ (v.s = \textsc{Executed} \ \wedge \\
& (\forall u \in R(v) : u.r_v = \textsc{Paused} \Rightarrow u.b_v \in \{v.p, \bot\}) \ \wedge \\
& (N(v) \neq \emptyset \Rightarrow \exists u \in R(v) : (u \in v.\mathit{bptr} \wedge u.r_v \neq \textsc{Paused}) \ \vee \\
& \qquad (u.r_v = \textsc{Paused} \wedge u.b_v = v.p))))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{actualBP}(v) &:= \{u \in R(v) \mid u.b_v \neq \bot\} \\
\mathit{pair}(v) &\equiv \deg(v) = 1 \ \wedge \ \forall u \in N(v) : u.k = 1 \\
\mathit{startUpper}_Q(v) &\equiv \neg(\mathit{backupConsistent}(v) \ \wedge \ \mathit{actualBP}(v) = \mathit{BP}(v)) \ \vee \\
& \quad (\neg \mathit{pair}(v) \ \wedge \ G_{\mathcal{A}}(v)) \ \vee \ (\mathit{pair}(v) \ \wedge \ \mathit{startReset}_Q(v))
\end{aligned}
$$

$$
\mathit{formerPeer}(v) := \begin{cases} u.\mathit{id} & \text{if } \exists!\, u \in N(v) : u.b_v \neq \bot \\ w.\mathit{id} & \text{if } \mathit{actualBP}(v) = \emptyset \ \wedge \ \exists!\, w \in N(v) : v.b_w \neq \bot \\ \bot & \text{otherwise} \end{cases}
$$

Figure 7.11: Predicates defined by upper layer

returned by $BP(v)$ (provided by the lower layer) or the values of the adjacent backups do not match the value of $v.p$. Furthermore, depending on whether v is part of a minor component of two nodes, a cycle should be started either if $\mathit{startReset}_Q(v)$ is true or node v is enabled with respect to $\mathcal{A}$. The Boolean predicate $\mathit{pair}(v)$ tests whether v is part of a minor component of two nodes. For this to be possible, we assume that an instance of protocol K as defined in Figure 7.8 exists on every node. Assuming that protocol K is disabled for all nodes, then $\mathit{pair}(v)$ is true if and only if a node v is part of a minor component of two nodes.

For minor components, the upper layer performs a reset of the primary variables of the nodes in such a way that algorithm $\mathcal{A}$ is disabled for the minor component after the reset. Section 7.3.7.2 explains how this reset is performed by procedure $\mathit{actionUpper}_Q$. The predicate $\mathit{startReset}_Q(v)$ decides whether the primary variable of v needs to be reset. Only for nodes that are not part of a minor component, regular moves of $\mathcal{A}$ are executed to eventually reach a legitimate primary configuration. Hence the truth of $G_{\mathcal{A}}(v)$ triggers another cycle if $\neg \mathit{pair}(v)$.

A cell C_v does not proceed with the transition Repaired $\rightarrow$ Executed and becomes blocked if any neighboring cells are not repaired (cf. Rule R2). This prevents cell C_v from executing moves of algorithm $\mathcal{A}$ that can read a

corrupted primary variable of neighboring cells and thus would lead to contamination. C_v remains blocked forever, unless all neighboring cells become repaired eventually. This is the case, as any non-repaired dialog-pausing cell does start a new cycle. For a dialog-paused cell $\neg \mathit{repaired}(v)$ implies $\neg \mathit{backupConsistent}(v)$ which in turn implies that $\mathit{startUpper}_Q(v)$ is true. Furthermore, a cell cannot become blocked as long as it is non-repaired. If a cell C_v is blocked, then it is dialog-consistent and $v.q = \textsc{Executed}$. Therefore, $v.s = \textsc{Repaired}$ and thus C_v is repaired by definition. Thus a circular wait between blocked cells can never occur, cf. Lemma 7.16.

Before the repair implemented by $\mathit{actionUpper}_Q$ and $\mathit{actionUpper}_{R_v}$ is explained in Section 7.3.7, we continue with the description of the middle layer which provides the satellite backups that are essential for the repair of corrupted primary variables to succeed.

7.3.6 Middle Layer

The implementation of the middle layer is again based on the cell technique as described in Section 7.3.3. It consists of the two procedures $\mathit{actionMiddle}_Q$ and $\mathit{actionMiddle}_{R_v}$ which are called by each (v, Q) and (u, R_v). The middle layer is responsible for maintaining the satellite backups of all $v.p$ with $\deg(v) \leq 2$. The procedures add more variables to the local states of (v, Q) and (u, R_v), $u \in N(v)$ respectively. A list of them and their domain is given in Figure 7.12. The variable $v.\mathit{sbptr}$ stores a set of pairs (w, u) as returned by $\mathit{SBP}(v)$. Recall from Section 7.3.4 that a pair $(w, u) \in \mathit{SBP}(v)$ indicates that (u, R_v) is to keep a satellite backup of $w.p$. The variable to store that satellite backup is called $u.\mathit{sb}_{v,w}$. Again, the value $\perp$ indicates that a certain satellite backup does not exist. Also, reading a satellite backup variable $u.\mathit{sb}_{v,w}$ of a non-existing instance (u, R_v) yields $\perp$ for notational convenience. When a new responding instance is spawned, it holds $u.\mathit{sb}_{v,w} = \perp$ for all v, w. It is assumed that (u, R_v) uses a suitable data structure to store only the variables $u.\mathit{sb}_{v,w} \neq \perp$, i.e., only these satellite backups are added to the size of the local state (u, R_v).

The implementation of $\mathit{actionMiddle}_Q$ and $\mathit{actionMiddle}_{R_v}$ is shown in Figure 7.12. During the transition $\textsc{Repaired} \to \textsc{Executed}$, procedure $\mathit{actionMiddle}_Q(v)$ sets $v.\mathit{sbptr}$ to the value of $\mathit{SBP}(v)$, which returns the current satellite backup distribution as computed by the lower layer. During the transition $\textsc{Executed} \to \textsc{Paused}$, $\mathit{actionMiddle}_{R_v}(u)$ then creates, deletes, or updates the satellite backup $u.\mathit{sb}_{v,w}$ depending on whether $(w, u) \in v.\mathit{sbptr}$. Note that it is not possible for (u, R_v) to access $w.p$. So in order to create or update the satellite backup $u.\mathit{sb}_{v,w}$, the value of $v.b_w$

Procedure $actionMiddle_Q(v)$

Nodes: v is the current node
Variables: $v.sbptr$: set of up to 2Δ pairs of node IDs

if $v.q = \text{EXECUTED}$ **then**
  $v.sbptr := SBP(v)$
end

Procedure $actionMiddle_{R_v}(u)$

Nodes: u is the current node, v is the center node
Variables: $u.sb_{v,w} \in \sigma_{\mathcal{A}} \cup \{\perp\}$
On Spawn: $u.sb_{v,w} := \perp$ for all v, w

if $v.q = \text{PAUSED}$ **then**
  foreach $(w, u) \notin v.sbptr$ **do** $u.sb_{v,w} := \perp$
  if $updateSB_{R_v}(u)$ **then**
    foreach $(w, u) \in v.sbptr$ **do** $u.sb_{v,w} := v.b_w$
  end
end

$$updateSB_{R_v}(u) \equiv (u \in v.bptr \wedge u.b_v = v.p) \vee (u \notin v.bptr \wedge u.b_v = \perp)$$

$$actualSBP(v) := \{(w, u) \mid u \in R(v) \wedge u.sb_{v,w} \neq \perp\}$$

$$startMiddle_Q(v) \equiv \neg(actualSBP(v) \subseteq SBP(v) = v.sbptr) \vee (\exists (w, u) \in SBP(v) : u.sb_{v,w} \neq v.b_w)$$

Figure 7.12: Implementation of the middle layer

instead of the value of $w.p$ is used. Note that temporarily $v.b_w \neq w.p$ may hold. However, the upper layer ensures that $v.b_w = w.p$ holds eventually for all nodes $w \in V$ with 2 neighbors or less. Consider the topology shown in Figure 7.4d. $b.sb_{a,v}$ is set to the value of $a.b_v$ which is eventually equal to $v.p$.

Procedure $actionMiddle_{R_v}$ updates satellite backups stored by (u, R_v) only if $updateSB_{R_v}(u)$ is true. The predicate $updateSB_{R_v}(u)$ is true if and only if the backup $u.b_v$ will not be modified by $actionUpper_{R_v}(u)$. As a result, any responding instance either updates adjacent or satellite backups

during a move, but not both. This makes it possible to serialize the moves of all responding instances, as shown in Section 7.4.4.

The middle layer must not delete, create, or update any satellite backups as long as a repair attempt by the upper layer is still pending. Recall that a cell becomes blocked if a neighboring cell is not repaired. Hence, the cell's dialog does not progress past position REPAIRED. Considering the topologies shown in Figure 7.4, this means that cell C_a does not modify any satellite backups of $v.p$ before C_v has attempted to repair $v.p$. The predicate $\mathit{startMiddle}_Q(v)$ is defined in such a way that cell C_v starts a new cycle when the distribution of the satellite backups does not match the one computed by the lower layer or when a satellite backup needs to be updated. Note that $\mathit{startMiddle}_Q(v)$ only tests whether $\mathit{actualSBP}(v)$ is a subset of $\mathit{SBP}(v)$ instead of testing whether the two are equal. As cell C_a with $a \in N(v)$ uses $a.b_v$ for creating the satellite backup of $v.p$, starting another cycle as long as $a.b_v = \bot$ would not create a satellite backup. The second half of $\mathit{startMiddle}_Q(a)$ ensures that a satellite backup of $v.p$ is eventually created as eventually $a.b_v = v.p$.

7.3.7 Fault Repair

This section describes the the repair mechanism which is implemented by $\mathit{actionUpper}_Q$ and $\mathit{actionUpper}_{R_v}$. It is assumed that a fault consisting of a topology change, a state corruption, or a combination of both occurs in a legitimate configuration of $\mathcal{A}_{FS}$. In Section 7.4, a legitimate configuration is defined such that it implies that adjacent backups and satellite backups for every node exist as described in Section 7.3.1, all cells are dialog-paused, and all nodes are disabled with respect to $\mathcal{A}$ prior to the fault. Let v be the node affected by state corruption. After the fault, a cell C_v becomes dialog-consistent at position PAUSED within a few moves and starts a new cycle if $\mathit{startUpper}_Q(v)$ is satisfied.

The transition from PAUSED to REPAIRED is the first transition executed by C_v. Also, all missing responding instances of cell C_v are spawned before the transition is completed. The repair of corruptions happens during this transition. In order to successfully perform the repair, the upper layer must cope with several different scenarios. They can be categorized as follows:

(1) v is part of a minor component neither prior nor after the fault.

(2) v is part of a minor component after the fault.

(3) v is part of a minor component prior to but not after the fault.

The cases are discussed one by one in Sections 7.3.7.1 to 7.3.7.3. Recall that a topology change is either an addition or removal of a single edge.

7.3.7.1 The Ordinary Case

Consider the case where v is not part of a minor component before or after the topology change. An edge removal can render up to two of four backups (if $\deg(v) \leq 2$ before the topology change, cf. Figure 7.4) or one of three backups (if $\deg(v) \geq 3$ before the topology change) inaccessible. However, in all cases, two backups remain accessible within the 2-hop neighborhood of v. A state corruption may now either corrupt $v.p$, corrupt one of the two remaining backups, or even delete the responding instance that holds one of the two remaining backups. The corruption of a backup also includes that the backup in question is set to $\perp$.

During the transition from PAUSED to REPAIRED, $actionUpper_{R_v}(u)$ is invoked for all $u \in N(v)$. Subsequently, $actionUpper_Q(v)$ is invoked. A backup, either adjacent or satellite, with a value equal to $v.p$ is called a *confirmation*. A confirmation indicates that the value of $v.p$ was not corrupted. Hence, one of the important tasks in the repair is to detect any confirmation within the 2-hop neighborhood of v. Confirming satellite backups are detected by $actionUpper_{R_v}(u)$ which sets $u.d_v$ to KEEP in line 3 if at least one confirming satellite backup exists. Procedure $actionUpper_Q(v)$, line 4, looks for confirmations within the 1-hop neighborhood or responding instances which have set $u.d_v = \text{KEEP}$. If these are discovered, the value of $v.p$ is not modified.

If no confirmations exist, then $v.p$ has been corrupted and two backups with an equal value exist within the 1-hop neighborhood of v or a responding instances of C_v. In line 4, $actionUpper_{R_v}(u)$ sets $u.d_v$ to UPDATE if a satellite backup $w.sb_{u,v} = u.b_v$, $w \in N(u)$ is detected by (u, R_v). $actionUpper_Q(v)$ updates $v.p$ with the value of $u.b_v$ if $u.d_v = \text{UPDATE}$ (cf. line 6) or if a second adjacent backup $w.b_v = u.b_v$, $u \neq w \in N(v)$ is detected (cf. line 5).

It is worth mentioning that a state corruption may also create a *forged backup* of $v.p$, i.e., a backup which did not exist before the state corruption. In that case, the state corruption affected a node other than v, and hence $v.p$ was not corrupted. The detection of forged backups is not required, as the repair mechanism will find a confirmation for the value of $v.p$. A forged backup may also be a confirmation.

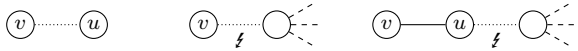


Figure 7.13: Topology changes that create minor components

7.3.7.2 Resetting Minor Components

Assume that node v is part of a minor component after the fault. This can have three reasons:

- Only a state corruption but no topology change occurred and v was part of a minor component prior to the fault.
- A topology change occurred and prior to it, v was an isolated node.
- A topology disconnected node v and at most one more node from the rest of the system.

These topology changes are depicted in Figure 7.13. Since v is part of a minor component, there are not enough backups available to restore the value of $v.p$ prior to a state corruption. It would be desirable to restore $v.p$ to a value such that after a constant number of rounds of any execution of $\mathcal{A}$, all subsequent configurations would satisfy $\mathcal{L}_{\mathcal{A}}$. Consider the case where the minor component that v is part of consists of 2 nodes, namely v and u. The question is whether $v.p$ should be overwritten with the value of $u.b_v$ or not. It might be possible to manually create a predicate for certain algorithms $\mathcal{A}$ that allows for testing whether overwriting or not overwriting $v.p$ with the value of $u.b_v$ would lead to the desired execution. However, we are not aware of a general method to create such a predicate. Hence, we suggest the following method which performs a reset of the local state of the primary variables of all nodes of the minor component.

In the case where v is isolated (i.e., $N(v) = \emptyset$), it is straightforward to reset the value of $v.p$ such that v is disabled with respect to $\mathcal{A}$ afterwards. Line 10 of $actionUpper_Q(v)$ does so during the transition from PAUSED to REPAIRED.

If node v has a neighbor u within the minor component, the reset is implemented as follows: Assuming that both cells C_v and C_u start a cycle, $u.d_v$ and $v.d_u$ are set to the value SINGLE in line 2 of $actionUpper_{R_v}(u)$ and $actionUpper_{R_u}(v)$. A value of $v.d_u = \text{SINGLE}$ indicates that u is the only neighbor of v. This allows v and u to determine that they form a minor component. In this case, both $actionUpper_Q(v)$ and $actionUpper_Q(u)$

Procedure $pairReset_Q(v, u)$

Nodes: v is the current node, u is a neighbor of v

1 **if** $v.b_u \neq u.p \wedge v.id < u.id \wedge v.b_u \neq \bot \wedge$
$\neg pairEnabled_{\mathcal{A}}(v : v.p, u : v.b_u)$ **then** Keep value of $v.p$
2 **else if** $pairEnabled_{\mathcal{A}}(u : u.p, v : v.p) \wedge u.b_v \neq \bot \wedge$
$\neg pairEnabled_{\mathcal{A}}(u : u.p, v : u.b_v)$ **then** $v.p := u.b_v$
3 **else if** $pairEnabled_{\mathcal{A}}(u : u.p, v : v.p)$ **then** $v.p := f_{\mathcal{A}}(v, u)$

$$startReset_Q(v) \equiv pairEnabled_{\mathcal{A}}(u : u.p, v : v.p) \wedge v.p \neq f_{\mathcal{A}}(v, u) \wedge (v.b_u = \bot \vee pairEnabled_{\mathcal{A}}(v : v.p, u : v.b_u))$$

where u denotes the only neighbor of v

Figure 7.14: Repairing corruptions in minor components

compute new values for $v.p$ and $u.p$ in line 3 by calling $pairReset(v, u)$ and $pairReset(u, v)$ respectively. The result is a legitimate primary configuration. Note that the regular execution of $\mathcal{A}$ would interfere with the reset mechanism. Thus, line 15 of $actionUpper_Q$ ensures that no moves of $\mathcal{A}$ are executed in minor components of two nodes.

The implementation of *pairReset* is given in Figure 7.14. It is in fact a modification of the procedure *pairRepair* as given in Figures 5.15 and 6.7. It assumes the existence of a function $f_{\mathcal{A}} : V \times V \to \sigma_{\mathcal{A}}$. The value of $f_{\mathcal{A}}(v, u)$ is assumed to satisfy $\neg pairEnabled_{\mathcal{A}}(v : f_{\mathcal{A}}(v, u), u : f_{\mathcal{A}}(u, v))$. Furthermore, $f_{\mathcal{A}}(v, u)$ is assumed to return the same value regardless of the local states of v and u, i.e., $f_{\mathcal{A}}(v, u)$ may only depend on static properties of v and u such as their identifiers. A possible implementation of $f_{\mathcal{A}}$ would compute the value $f_{\mathcal{A}}(v, u)$ as follows: For a virtual system with the topology $V = v, u$ and $E = (v, u)$, $v.p$ and $u.p$ are initialized with a constant value $x \in \sigma_{\mathcal{A}}$. Then, an execution of $\mathcal{A}$ starting in this initial configuration is performed: Alternately, starting with the node with the highest identifier, nodes v and u execute **if** $G_{\mathcal{A}}$ **then** $S_{\mathcal{A}}$. When $\mathcal{A}$ has terminated, then the value of $v.p$ is the return value of $f_{\mathcal{A}}(v, u)$.

Lines 1 and 2 of *pairReset* first attempt to repair the state corruption under the assumption that there was no topology change. The method used has been discussed in Section 5.8.4.3. If this fails, then line 3 performs the reset of $v.p$ to the value of $f_{\mathcal{A}}(v, u)$. In the latter case, it is assumed that $u.p$ has already been or will be overwritten with the value of $f_{\mathcal{A}}(u, v)$, unless both nodes are already disabled with respect to $\mathcal{A}$.

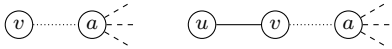


Figure 7.15: A minor component is connected to another component

For minor components of two nodes, the predicate $startReset_Q(v)$ controls whether a cycle is started (cf. the definition of $startUpper_Q(v)$ in Figure 7.11). Both cells C_v and C_u should start a cycle if v or u is enabled with respect to $\mathcal{A}$, i.e., if $pairEnabled_{\mathcal{A}}(v : v.p, u : u.p)$, and if a reset of $v.p$ or $u.p$ has not been performed yet, i.e., $v.p \neq f_{\mathcal{A}}(v, u)$ or $u.p \neq f_{\mathcal{A}}(u, v)$ respectively. However, $startReset_Q(v)$ does not force the start of a cycle by C_v if it is clear that C_u starts a cycle and will overwrite $u.p$ with $v.b_u$ in line 2 of $pairReset(u, v)$. This is the case if $v.b_u \neq \bot \wedge \neg pairEnabled_{\mathcal{A}}(v : v.p, u : v.b_u)$.

Note that the definition of $startUpper_Q(v)$ uses the value of $pair(v)$ to decide between the of $G_{\mathcal{A}}(v)$ or $startReset_Q(v)$. The value of $pair(v)$ depends on the value of $u.k$ provided by (u, K). The model as defined in Chapter 2 does not guarantee in any case that (u, K) is selected eventually by the scheduler so that the variable $u.k$ is updated. Hence, $pair(v)$ may remain false indefinitely. In particular, a scenario may exist where node v is enabled with respect to $\mathcal{A}$ even though it has already performed the reset to $v.p = f_{\mathcal{A}}(v, u)$. Under the unfair scheduler, cell C_v may thus execute an indefinite number of cycles since $startUpper_Q(v)$ is true indefinitely. Hence, we make the assumption that an enabled instance (u, K) makes a move before the second move of (u, R_v). This guarantees that all variables $u.k$ with $u \in N(v)$ are up-to-date after one cycle of C_v. This assumption can easily be satisfied even under the unfair scheduler by applying the transformation given in Section 2.7 to $\mathcal{A}_{FS}$.

7.3.7.3 Detecting Former Minor Components

The addition of an edge which connects a minor component to another component of the system is equally challenging to the opposite case discussed in the previous section. This is due to the fact that one cannot establish the necessary redundancy to reverse a state corruption within a minor component.

For nodes that have been part of a minor component prior to the addition of an edge, at most one backup exists prior to the topology change. In particular, if a node has been part of a minor component, then the state

corruption may delete the only backup in existence, or it may even add a forged backup. Hence, it is impossible to determine the original value of the primary variable of such nodes prior to a state corruption based on the values of the backups. Instead, the upper layer detects the edge that has been added by the topology change in order to identify the former minor component. For minor components of two nodes, the technique described in Section 5.8.4.3 and implemented in *pairReset* is used. For a minor component of a single node, a reset mechanism is implemented.

Now consider the topology as shown in Figure 7.15 on the right. Nodes v and u formed a minor component before the topology change that added the edge (v, a). The case where nodes u and v were not affected by the state corruption can be ignored. In that case, $u.b_v$ is a confirmation of $v.p$ and $v.b_u$ is a confirmation of $u.p$ and so neither C_v nor C_u change their primary variables during repair. Hence, assume that either u or v was affected by a state corruption. The following logic is used to identify node u among v's neighbors: If v sees exactly one adjacent backup of its own primary variable, then it must be $u.b_v$. It is not relevant whether $u.b_v$ was corrupted. The mere existence of $u.b_v$ identifies node u.

However, the state corruption may have deleted the backup $u.b_v$. In that case, none of v's neighbors stores a backup of $v.p$. Since node v was not affected by the state corruption, v holds a backup of exactly one of its neighbors, namely node u and not node a. This detection is implemented by the function $\mathit{formerPeer}(v)$ as given in Figure 7.11, where the quantifier $\exists!$ denotes "there exists exactly one". For the example shown in Figure 7.15, $\mathit{formerPeer}(v)$ returns the ID of node u.

Note that the identification of u is based on the existence and location of backups of $v.p$ and $a.p$. Potentially, cell C_a may create the backup $v.b_a$ and C_v may create $a.b_v$. Afterwards $\mathit{formerPeer}(v)$ ceases to correctly identify node u. Recall that cells can't complete the transition Repaired $\rightarrow$ Executed unless all neighboring cells are repaired. Due to this synchronization mechanism between neighboring cells, C_v cannot create the backup $a.b_v$ unless cell C_u is repaired and C_a cannot create $v.b_a$ unless C_v is repaired.

Consider the case where $u.b_v$ is not deleted by the state corruption. The creation of $a.b_v$ does not happen before C_u and C_v are both repaired. Hence, the value of $\mathit{formerPeer}(v)$ remains correct as long as needed. Now consider the case where $u.b_v$ is deleted by the state corruption. Obviously, C_v is not repaired as long as $u.b_v = \perp$ and C_v has not reached position Repaired. However, when C_v reaches position Repaired the cell becomes repaired and the value of $\mathit{formerPeer}(v)$ may potentially change if C_a creates $v.b_a$. Hence,

before C_v becomes repaired, line 2 of $actionUpper_Q(v)$ saves the return value of $formerPeer(v)$ in the variable $v.fp$ for later use by $actionUpper_{R_u}(v)$ in line 5. Note that C_v does not re-create the deleted backup $u.b_v$ until C_u is repaired. Hence, $actionUpper_{R_u}(v)$ uses the value $v.fp$ if C_v has become repaired and no backups of $v.p$ exist. Otherwise, the value of $formerPeer(v)$ is used directly in line 6 of $actionUpper_{R_u}(v)$.

During the transition from PAUSED to REPAIRED, $actionUpper_{R_u}(v)$ sets the variable $v.d_u$ to MINOR (either line 5 or line 6) and $actionUpper_{R_v}(u)$ sets $u.d_v$ to SINGLE (line 2). The value MINOR indicates that v currently has more than one neighbor, but that u was identified as the node that formed a minor component with v before the topology change. The value SINGLE indicates that u only has one neighbor, namely node v. Unless a confirmation was found, $actionUpper_Q(v)$ calls $pairReset(v, u)$ in line 7 and $actionUpper_Q(u)$ calls $pairReset(u, v)$ in line 8. Note that either the assignment $u.p := v.b_u$ or $v.p := u.b_v$ yields the configuration such that both nodes would be disabled with respect to $\mathcal{A}$ prior to the topology change. Since only one node's state was corrupted, it is sufficient that one node overwrites its primary variable with the backup. Thus, line 3 of *pairReset* is never used in this case.

It remains to discuss the case where node v formed a minor component of a single node before the topology change as depicted in Figure 7.15 on the left. Let the edge (v, a) be the one added by the topology change. Furthermore, it is assumed that node a has at least one more neighbor $b \neq u$. Without loss of generality it is assumed that b is a neighbor of v that held a backup of a. Forged confirmations for the value of $v.p$ may exist. However, in that case, $v.p$ was not corrupted and the value of $v.p$ is not altered by $actionUpper_Q(v)$. A non-confirming forged backup can exist. However, forged backups on two different nodes would be needed in order for $actionUpper_{R_v}(a)$ to set $a.d_v$ to UPDATE. It remains to show that $a.d_v$ is not set to MINOR by $actionUpper_{R_v}(a)$. If the state corruption did not delete $b.b_a$ then $formerPeer(a)$ returns either b or $\perp$. Also, cell C_a cannot move its backup away from b before C_v is repaired. If the state corruption deletes $b.b_a$, then the state corruption did not affect node a or v. From that it follows that $a.b_v = \perp$ and $v.b_a = \perp$. That implies that $formerPeer(a) \neq v$ prior to C_v being repaired. It follows that $actionUpper_Q(v)$ resets $v.p$ in line 10 to a value such that node v would be disabled with respect to $\mathcal{A}$ prior to the topology change.

7.4 Analysis

Let $\mathcal{A}_L$ denote the algorithm that is used as the lower layer. It is assumed to be silent and self-stabilizing with respect to $\mathcal{L}_{\mathcal{A}_L}$, where $\mathcal{L}_{\mathcal{A}_L}$ shall imply that the computed backup placement satisfies the requirements as given in Sections 7.3.1 and 7.3.4. Without loss of generality, we assume $K \in \mathcal{A}_L(v)$ for all $v \in V$ and that $\mathcal{L}_{\mathcal{A}_L}$ implies $v.k = k(v)$ for all $v \in V$. We then define the output of the transformation as $\mathcal{A}_{FS} = \mathcal{A}_L \triangleright \mathcal{A}'$ where $\mathcal{A}'(v) = \{M, Q\}$ for all $v \in V$. Note that $\mathcal{A}'$ does not include any instances of protocol R_v since they are spawned at runtime by the instances of M.

Recall that $\mathcal{A}$, the algorithm that is input to the transformation, is assumed to be silent and self-stabilizing with respect to the Boolean predicate $\mathcal{L}_{\mathcal{A}}$. Furthermore, it is assumed to be Λ-super-stabilizing with respect to $\mathcal{S}_{\mathcal{A}}$, where Λ denotes the class of topology changes which describes the removal or addition of a single edge. A combination of both is not considered. We define

$$\begin{aligned}\mathcal{L}_{\mathcal{A}_{FS}} \equiv \forall v \in V : \neg G_{\mathcal{A}_L}(v) \wedge K(v) = \{R_u \mid u \in N(v)\} \wedge \\ \mathit{dialogPaused}(v) \wedge \neg \mathit{startUpper}_Q(v) \wedge \neg \mathit{startMiddle}_Q(v)\end{aligned}$$

where $G_{\mathcal{A}_L}(v)$ denotes the predicate that is true if and only if node v is enabled with respect to $\mathcal{A}_L$. We proceed to show that $\mathcal{A}_{FS}$ is silent, self-stabilizing with respect to $\mathcal{L}_{\mathcal{A}_{FS}}$, provides fault-containing self-stabilization with respect to $\mathcal{L}_{\mathcal{A}}$, and is fault-containing Λ-super-stabilizing with respect to $\mathcal{S}_{\mathcal{A}}$. Note that $\mathcal{L}_{\mathcal{A}_{FS}}$ implies that $R(v) = N(v)$, that all adjacent and satellite backups of $v.p$ are equal to $v.p$, and that the placement of the backups is as computed by the lower layer, i.e., $\mathit{actualBP}(v) = \mathit{BP}(v)$ and $\mathit{actualSBP}(v) = \mathit{SBP}(v)$. Also, if $\mathcal{L}_{\mathcal{A}_{FS}}$ is true then the lower layer has terminated.

Theorem 7.5. *The transformation is correct, i.e., $\mathcal{L}_{\mathcal{A}_{FS}}$ implies $\mathcal{L}_{\mathcal{A}}$ for all configurations.*

Proof. Let $(E, c) \in \Sigma^{ext}_{\mathcal{A}_{FS}}$ denote a configuration that satisfies $\mathcal{L}_{\mathcal{A}_{FS}}$. We show that all nodes are disabled with respect to $\mathcal{A}$ in (E, c). Since $\mathcal{A}$ is self-stabilizing, this implies that $\mathcal{L}_{\mathcal{A}}$ is satisfied in (E, c).

$\mathcal{L}_{\mathcal{A}_{FS}}$ implies that $\neg G_{\mathcal{A}_L}(v)$ and thus $v.k = k(v)$ for all $v \in V$. Hence, $\mathit{pair}(v)$ is true if and only if v is part of a minor component of two nodes. For all nodes that are not part of such a minor component, $\neg \mathit{startUpper}_Q(v)$ thus implies that $\neg G_{\mathcal{A}}(v)$.

Let nodes v and u form a minor component of two nodes in (E, c). Then $\neg \mathit{startUpper}_Q(v)$ and $\neg \mathit{startUpper}_Q(u)$ in (E, c). From that it follows that C_v and C_u are backup-consistent and that $\neg \mathit{startReset}_Q(v)$ and $\neg \mathit{startReset}_Q(u)$ in (E, c). Hence, $v.b_u = u.p$ and $u.b_v = v.p$. Assume that $G_{\mathcal{A}}(v)$ or $G_{\mathcal{A}}(u)$. Then $\mathit{pairEnabled}_{\mathcal{A}}(v : v.p, u : u.p)$, $\mathit{pairEnabled}_{\mathcal{A}}(v : v.p, u : v.b_u)$, $\mathit{pairEnabled}_{\mathcal{A}}(u : u.p, v : v.p)$, and $\mathit{pairEnabled}_{\mathcal{A}}(u : u.p, v : u.b_v)$ are satisfied. Furthermore, $v.p \neq f(v, u)$ or $u.p \neq f(u, v)$. Hence, either $\mathit{startReset}_Q(v)$ or $\mathit{startReset}_Q(u)$ is satisfied. This is a contradiction to $\neg \mathit{startUpper}_Q(v)$ and $\neg \mathit{startUpper}_Q(u)$ in (E, c). Hence, both nodes v and u are disabled with respect to $\mathcal{A}$ in (E, v). □

Since the exact memory consumption per node depends on the backup placement computed by the lower layer, we give an upper bound on the average space consumed per node instead.

Theorem 7.6 (Stabilization Space)**.** *In a legitimate configuration, the average space required per node is $\mathcal{O}(S_{\mathcal{A}_L} + S_{\mathcal{A}} + \frac{m}{n} \log n)$, where $S_{\mathcal{A}_L}$ and $S_{\mathcal{A}}$ denote the average space required per node by the lower layer and algorithm $\mathcal{A}$ respectively.*

Proof. Each primary variable and the four backups thereof account for $\mathcal{O}(S_{\mathcal{A}})$ bits per node. The variables of instances of R_v excluding the backups consume $\mathcal{O}(\log n)$ bits per edge and hence $\mathcal{O}(\frac{m}{n} \log n)$ bits on average per node. The variables of protocol Q (excluding the primary variable and $v.\mathit{sbptr}$) consume $\mathcal{O}(\log n)$ bits per node. As each node $v \in V$ creates at most 2 satellite backups, at most two entries of the form $(v, x) \in u.\mathit{sbptr}$ for at most two nodes $u \in N(v)$ are created. Each entry consumes $\mathcal{O}(\log n)$ bits. □

Note that some initial configurations can cause a larger memory consumption during an initial phase. This is due to instances $R_v \in K(u)$ with $v \notin N(u)$ and unnecessary satellite backups. These are deleted within a constant number of rounds and afterwards the space requirements satisfy the given bound.

Theorem 7.7 (Self-Stabilization)**.** *$\mathcal{A}_{FS}$ is silent and self-stabilizing with a stabilization time of $\mathcal{O}(1) + \max\{T_{\mathcal{A}_L}, 6T_{\mathcal{A}}\}$ rounds, where $T_{\mathcal{A}_L}$ and $T_{\mathcal{A}}$ denote the termination time of the lower layer and $\mathcal{A}$ in rounds respectively.*

Theorem 7.8 (Fault-Containing Self-Stabilization)**.** *$\mathcal{A}_{FS}$ is fault-containing self-stabilizing with respect to $\mathcal{L}_{\mathcal{A}}$ with a contamination number of $\mathcal{O}(1)$, a containment time of $\mathcal{O}(1)$ rounds, and a fault-gap of $\mathcal{O}(1) + T_{\mathcal{A}_L}$ rounds,*

where $T_{\mathcal{A}_L}$ denotes the stabilization time of the lower layer for 1-faulty initial configurations in rounds.

Theorem 7.9 (Fault-Containing Super-Stabilization)**.** *$\mathcal{A}_{FS}$ is fault-containing Λ-super-stabilizing with respect to $\mathcal{S}_{\mathcal{A}}$ with a containment time of $\mathcal{O}(1)$ rounds and a fault-gap of $\mathcal{O}(1) + \max\{T_{\mathcal{A}_L}, 6T_{\mathcal{A}}\}$ rounds, where $T_{\mathcal{A}_L}$ denotes the termination time of the lower layer for $(\Lambda, 1)$-faulty initial configurations and $T_{\mathcal{A}}$ the fault-gap of $\mathcal{A}$ for Λ-faulty initial configurations in rounds respectively.*

The detailed proofs of Theorems 7.7 to 7.9 follow. The proofs assume that in each step at most one instance per cell is selected. That greatly simplifies the proofs. This assumption is clearly true for the central scheduler. In Section 5.8.5.3 it is shown that for any step of $\mathcal{A}_{FS}$ under the distributed scheduler, a partial serialization exists such that in each step of the serialization at most one instance per cell makes a move. Furthermore, the proofs assume that the scheduler selects any enabled instance (v, K) before the second move of a responding instance (v, R_u) with $u \in N(v)$, cf. Section 7.3.7.2. The partial serialization preserves this assumption for any execution.

Note that $\mathcal{A}_{PR}$, as described in Section 7.3.4, terminates within two rounds, regardless of the initial configuration. If $\mathcal{A}_{PR}$ is used for the lower layer, then $T_{\mathcal{A}_L}$ in Theorems 7.7 to 7.9 can be assumed to be 2.

7.4.1 Proof of Self-Stabilization

The case where minor components exist is excluded from the proofs in this section. This case is handled as described in Section 7.3.7.2. For the remainder of this section it is assumed that all connected components consist of at least three nodes each.

It is said that cell C_v is enabled if (v, Q) is enabled, (u, R_v) with $u \in R(v)$ is enabled, or $v.s = \textsc{Paused}$ and $R_v \notin K(u)$ with $u \in N(v)$. It is said that C_v has made a move if any of its instances (v, Q) or (u, R_v), $u \in R(v)$ have made a move or if a responding instance has been spawned. Furthermore, a dialog-consistent cell C_v is called *unacknowledged* if $u.r_v = v.s$ for all $u \in R(v)$. In particular, a dialog-paused cell is also regarded to be unacknowledged.

Lemma 7.10. *A non-dialog-consistent cell is enabled and becomes dialog-consistent and unacknowledged at position* Paused *after at most 2Δ moves. These moves happen within at most 2 rounds. dialogConsistent(v) is stable in any execution of $\mathcal{A}_{FS}$.*

Proof. Let C_v denote a non-dialog-consistent cell.

Case a) $v.s = v.q = \textsc{Paused}$. Since C_v is not dialog-consistent, the response-variable of at least one responding instance is not equal to PAUSED or the cell is non-operational. If the cell is non-operational, at least one responding instance is spawned within the first round. Its response-variable is initialized with PAUSED. Afterwards, C_v is dialog-paused. If C_v is operational, then some responding instance (u, R_v) exists with $u.r_v \neq \textsc{Paused}$. Every such responding instance is enabled and resets $u.r_v$ to PAUSED via Rule R1. All other responding instances are disabled. After all response-variables have been reset, C_v is dialog-paused (and thus dialog-consistent and unacknowledged). Note that until that happens, new responding instances may be spawned, and their response-variables are initialized with PAUSED. These responding instances are disabled after being spawned. The up to Δ moves happen within one round.

Case b) *validQuery*(v). Rule Q1 of (v, Q) is enabled. If cell C_v is non-operational and $v.s = \textsc{Paused}$, then a responding instance (u, R_v) may be spawned before (v, Q) makes a move. That renders C_v dialog-consistent and unacknowledged since $u.r_v$ is initialized with PAUSED after one move that happens within one round. Otherwise, (v, Q) resets $v.s$ and $v.q$ to PAUSED via Rule Q1 within one round. Note that depending on the value of the response-variables, up to $\Delta - 1$ responding instances may execute Rule R2 before (v, Q) makes a move. After at most Δ moves within one round, case a) applies.

Case c) $\neg(\textit{validQuery}(v) \vee v.s = v.q = \textsc{Paused})$. All responding instances of C_v are disabled. Rule Q1 of (v, Q) is enabled. Within one round, (v, Q) sets $v.s$ and $v.q$ to PAUSED. If cell v is not yet dialog-paused, case a) applies.

Let C_v denote a dialog-consistent cell. Rule Q1 of (v, Q) as well as Rule R1 of all (u, R_v), $u \in R(v)$ are disabled since C_v is dialog-consistent. If Rule Q2 is enabled, then $v.s = v.q = \textsc{Paused}$ and $w.r_v = v.s = \textsc{Paused}$ for all $w \in R(v)$. After changing $v.q$ to REPAIRED, dialog-consistency still holds. If Rule Q3 is enabled, then $w.r_v = v.q$ for all $w \in R(v)$. After setting $v.s := v.q$, it holds $w.r_v = v.s$ for all $w \in N(v)$. $v.q$ is updated in such a way that $(v.s, v.q)$ is a valid pause or query. Hence C_v remains dialog-consistent. If a responding instance (u, R_v) executes Rule R2, then $(v.s, v.q)$ is a valid query and $u.r_v$ is set to $v.q$. Hence C_v remains dialog-consistent. If a new responding instances (u, R_v) is spawned, it holds $v.s = \textsc{Paused}$. Since $u.r_v$ is initialized with PAUSED, it holds $u.r_v = v.s$. Thus, C_v remains dialog-consistent. □

For a dialog-consistent cell C_v, it is said that it starts a new cycle if (v, Q) sets $(v.s, v.q) := (\textsc{Paused}, \textsc{Repaired})$. This happens either via Rule Q2 or Rule Q3. A dialog-consistent cell C_v completes the transition to a position x with the move that sets $v.s := x$. This is always a result of an execution of Rule Q3 of (v, Q). Cell C_v completes a cycle if C_v completes the transition to position Paused. In particular, the same move of (v, Q) may both complete and start a cycle if $\mathit{startUpper}_Q(v)$ or $\mathit{startMiddle}_Q(v)$.

Lemma 7.11. *A dialog-consistent cell at position* Paused *starts and completes a cycle within at most* $4\Delta + 3$ *moves. Unless the cell is blocked at some point, these moves happen within at most* 7 *rounds.*

Proof. Let C_v denote a dialog-consistent cell. At least one responding instance of C_v, since C_v is dialog-consistent. Also, assume that all cells neighboring C_v are repaired and thus that C_v does not become blocked.

Case a) Assume that $v.s = v.q =$ Paused and that $\mathit{startUpper}_Q(v)$ or $\mathit{startMiddle}_Q(v)$ is satisfied. Within the first round, (v, Q) executes Rule Q2 and sets $v.q :=$ Repaired and all missing responding instances are spawned since $v.s =$ Paused.

Case b) Assume that $(v.s, v.q) = (\textsc{Paused}, \textsc{Repaired})$. In that case, all missing responding instances are spawned since $v.s =$ Paused within the first round.

In both cases, the response-variables are initialized with Paused for any spawned instance. The execution continues as follows: By the end of the second round, all responding instances have acknowledged the query by (v, Q) via Rule R2 and C_v is dialog-acknowledged. Before the end of the third round, (v, Q) has executed Rule Q3 and sets $(v.s, v.q) := (\textsc{Repaired}, \textsc{Executed})$. By the end of the fourth round, all responding instances acknowledge the query for a transition to Executed. In the fifth round, (v, Q) sets $(v.s, v.q)$ to $(\textsc{Executed}, \textsc{Paused})$. The acknowledgments follow in the sixth round. (v, Q) sets $v.s :=$ Paused in the seventh round. □

Corollary 7.12. *A dialog-consistent cell at position* Paused *completes the transition to position* x *within at most* $(x + 1)(\Delta + 1) - 1$ *moves. If* $x \leq$ Repaired *or the cell does not become blocked, these moves happen within at most* $1 + 2x$ *rounds.*

Corollary 7.13. *A dialog-consistent cell at position* $x \neq$ Paused *completes a cycle within at most* $(3-x)(\Delta+1)$ *moves. Unless the cell becomes blocked, these moves happen within at most* $2(3 - x)$ *rounds.*

Corollary 7.14. *A non-dialog-paused cell is enabled unless it is blocked.*

Corollary 7.15. *A dialog-consistent cell C_v at position* EXECUTED *finishes its current cycle, starts a new cycle, and reaches position* EXECUTED *within at most* 6 *rounds, assuming that C_v does not become blocked and that $startUpper_Q(v) \vee startMiddle_Q(v)$ is satisfied continuously.*

Note that it takes a cell C_v only 6 rounds for a cycle from EXECUTED to EXECUTED, as Rule Q3 allows C_v to complete and start a cycle with one move.

Lemma 7.16. *A non-repaired cell is enabled and becomes repaired after at most $4\Delta + 3$ moves. These moves happen within* 7 *rounds. The predicate repaired(v) is stable in any execution of $\mathcal{A}_{FS}$.*

Proof. Let c_0 be a configuration such that cell C_v is not repaired.

Case a) v is dialog-consistent and $v.s =$ PAUSED in c_0. C_v is not repaired and thus not backup-consistent. Hence $startUpper_Q(v)$ is satisfied. By Corollary 7.12, C_v reaches position REPAIRED within $2\Delta + 1$ moves or 3 rounds and thereby becomes repaired.

Case b) v is dialog-consistent and $v.s \neq$ PAUSED in c_0. Since C_v is not repaired it holds $v.s \neq$ REPAIRED. By Corollary 7.13, it takes C_v at most $2\Delta + 2$ moves or 4 rounds to finish the current cycle. Now case a) applies.

Case c) C_v is not dialog-consistent in c_0. By Lemma 7.10 v becomes dialog-consistent with $v.s =$ PAUSED in 2Δ moves or 2 rounds. Now case a) applies.

Let C_v denote a repaired cell. Let (u, R_v), $u \in R(v)$ denote a responding instance of C_v. C_v is dialog-consistent and by Lemma 7.10 remains dialog-consistent in any execution. Thus Rule Q1 of (v, Q) as well as Rule R1 of (u, R_v) are disabled.

Case a) $v.s =$ PAUSED. It follows $v.q \in \{$PAUSED, REPAIRED$\}$ and that C_v is backup-consistent. A move of (u, R_v) does not delete, create or update any backups and thus does not affect backup-consistency of C_v. A move (v, Q) (either Rule Q2 or Rule Q3) does not change $v.p$, unless $v.s =$ REPAIRED afterwards. In any case, C_v remains repaired.

Case b) $v.s =$ REPAIRED. It follows that $v.q =$ EXECUTED. C_v remains repaired after a move of (u, R_v) since $v.s =$ REPAIRED. Rule Q2 of (v, Q) is disabled, since C_v is not dialog-paused. A move by (v, Q) (Rule Q3) may change $v.p$ and sets $v.q$ to EXECUTED. In addition, $v.bptr$ is updated with the value of $BP(v)$, which is guaranteed to contain at least one $w \in R(v)$. Since $w.r_v =$ EXECUTED holds for all responding instances (w, R_v), $w \in R(v)$, prior to the move of (v, Q) and thus afterwards, C_v remains repaired.

Case c) $v.s = \text{EXECUTED}$. It follows that $v.q = \text{PAUSED}$. Rule Q2 of (v, Q) is disabled, since C_v is not dialog-paused. If Rule Q3 of (v, Q) is enabled, this implies that C_v is dialog-acknowledged. It follows that $u.r_v = \text{PAUSED}$ for all $u \in R(v)$. From C_v being repaired it follows that $u.b_v = v.p$ for at least one $u \in R(v)$ and $u.b_v \in \{v.p, \bot\}$ for all other $u \in R(v)$. Hence, C_v is backup-consistent and remains backup-consistent under the move of (v, Q) which does not modify $v.p$ if $v.q = \text{PAUSED}$. If (u, R_v) is enabled (Rule R2), then $u.r_v = \text{EXECUTED}$. From C_v being repaired it follows that $w \in R(v)$ with $w.b_v = v.p$ exists or $u \in v.bptr$. In the former case, C_v remains repaired regardless of whether $u.b_v = \bot$ or $u.b_v = v.p$ after the move of (u, R_v). In the latter case, $u.b_v = v.p$ holds after the move of (u, R_v). Hence, C_v remains repaired. □

Lemma 7.17 (Progress)**.** *Let C_v denote a dialog-consistent, unacknowledged, and repaired cell at position* PAUSED*. For any execution, any modification of $v.p$ is a result of moves of $\mathcal{A}$ during the transition* REPAIRED $\rightarrow$ EXECUTED*. If $\neg pair(v)$ and all cells neighboring C_v are repaired, then (v, Q) executes $S_\mathcal{A}$ within these 5 rounds provided that $G_\mathcal{A}(v)$ is true continuously for the next 5 rounds.*

Proof. From $v.s = \text{PAUSED}$ and C_v being repaired it follows that C_v is backup-consistent. Thus, at least one confirmation $u.b_v = v.p$, $u \in R(v)$ exists. Furthermore, $u.r_v = \text{PAUSED}$ holds for all $u \in R(v)$. Also, the response variable of each responding instance that is spawned is initialized with PAUSED. The definition of $dialogAcknowledged(v)$ makes sure that $R(v) = N(v)$ before the transition to REPAIRED. In particular, every responding instance executes Rule R2, which updates the value of $u.d_v$, before C_v becomes dialog-acknowledged. The assumption that v is not part of a minor-component yields that either $\deg(v) > 1$ or $u.d_v \neq \text{SINGLE}$ for all $u \in N(v)$. From this and the fact that at least one confirmation exists, it follows that $actionUpper_Q(v)$ does not modify $v.p$ or call $pairReset$ during the transition to REPAIRED.

Assume $\neg pair(v)$ and that all cells neighboring C_v be repaired. Then $startUpper_Q(v)$ is true and C_v reaches position EXECUTED within 5 rounds by Corollary 7.12. From $\deg(v) > 1$ or $u.d_v \neq \text{SINGLE}$ and the assumption that $G_\mathcal{A}(v)$ is true it follows that $actionUpper_Q(v)$ executes $S_\mathcal{A}$ during the transition to EXECUTED. □

Theorem 7.18 (Partial Correctness)**.** *If all nodes are disabled, then $\mathcal{L}_{\mathcal{A}_{FS}}$ is satisfied.*

Proof. From Lemma 7.16 it follows that all cells are repaired. Hence, no cell can be blocked. From Corollary 7.14 it follows that all cells are dialog-paused. From (v, M) being disabled it follows that $K(v) = \{R_u \mid u \in N(v)\}$ for all $v \in V$. Furthermore, $\neg G_{\mathcal{A}_L}(v)$ for all $v \in V$. From Rule Q2 of (v, Q) being disabled it follows that $\neg \mathit{startUpper}_Q(v)$ and $\neg \mathit{startMiddle}_Q(v)$ for all $v \in V$. Hence, $\mathcal{L}_{\mathcal{A}_{FS}}$ is satisfied. □

Theorem 7.19 (Closure)**.** *$\mathcal{L}_{\mathcal{A}_{FS}}$ is stable in any execution of $\mathcal{A}_{FS}$.*

Proof. In a legitimate configuration, $\neg G_{\mathcal{A}_L}(v)$, $\mathit{dialogPaused}(v)$, $\neg \mathit{startUpper}_Q(v)$, $\neg \mathit{startMiddle}_Q(v)$, and $K(v) = \{R_u \mid u \in N(v)\}$ for all $v \in V$. Hence, the lower layer and all cells are disabled. Also, all instances of M are disabled. Hence, all nodes are disabled. □

Theorem 7.20 (Termination)**.** *After a finite number of moves, a configuration is reached in which all nodes are disabled.*

Proof. Since $\mathcal{A}_{FS}$ is a collateral composition of $\mathcal{A}_L$ and $\mathcal{A}'$ with $\mathcal{A}'(v) = \{M, Q\}$ for all $v \in V$, it suffices to show $\mathcal{A}'$ is strongly silent by Theorem 4.10. Hence, the execution of $\mathcal{A}_{FS}$ is assumed not to contain any moves of instances of $\mathcal{A}_L$. Hence, the output of $\mathcal{A}_L$ does not change. Protocol M makes at most 1 move per node. It remains to show that each cell makes a finite number of moves, regardless of the output of $\mathcal{A}_L$.

Let C_v denote a cell. By Lemmas 7.10 and 7.16, C_v becomes dialog-consistent and repaired within a finite number of moves. By Lemma 7.11 or Corollary 7.13, C_v becomes dialog-paused and repaired within a finite number of moves. Now, Lemma 7.17 yields that any future modification of $v.p$ is the result of moves of $\mathcal{A}$. In conclusion, the number of modifications of a primary variable that are not a result of a move of are finite. Such modification may reset the stabilization progress of $\mathcal{A}$ finitely often. In spite of that, the total number of moves of protocol $\mathcal{A}$ is finite since $\mathcal{A}$ is self-stabilization and silent. Thus the total number of modifications of each primary variable (either by moves of $\mathcal{A}$ or not) is finite.

Let C_v denote a dialog-consistent cell at position PAUSED that has started and completed at least one cycle. Then the backups (adjacent and satellite) have been moved to the locations selected by the lower layer, i.e., $\mathit{actualBP} = \mathit{BP}(v)$ and $\mathit{actualSBP} \subseteq= \mathit{SBP}(v)$. Furthermore, C_v is backup-consistent and $R(v) = N(v)$ by definition of $\mathit{dialogAcknowledged}(v)$. By definition of $\mathit{startUpper}_Q(v)$ and $\mathit{startMiddle}_Q(v)$, cell C_v starts a new cycle only for one of the following reasons:

(1) $pair(v) \wedge startReset_Q(v)$

(2) $\neg pair(v) \wedge G_{\mathcal{A}}(v)$

(3) satellite backups are to be updated

If C_v starts a cycle for reason (1), then $u.k \neq k(v)$ with $u \in N(v)$ as v is assumed not to be part of a minor component. Hence, the execution only contains a finite number of moves of C_v, as the dialog of C_v cannot progress unless the execution contains a move of (u, K). By assumption the execution does not contain any moves of $\mathcal{A}_L$.

If C_v starts a new cycle for reason (2), then by Lemma 7.17 C_v executes moves of $\mathcal{A}$ unless $G_{\mathcal{A}}(v)$ has become false due to a modification of a primary variable $u.p$, $u \in N(v)$. Hence, at least one modification of a primary variable happens per cycle of C_v started for reason (2). Since the total number of modifications of any primary variable is finite, the number of cycles that C_v starts for reason (2) is also finite.

Of the cycles started for reason (3), we count only those where $v.p$ is not modified. As previously shown, a modification of $v.p$ occurs only finitely often. Only if $v.p$ is not modified, then the C_v is still backup consistent during the transition from Executed to Paused. Only then, the satellite backups stored by responding instances of C_v are updated. Note that in order for C_v to start a new cycle for reason (3), a backup $v.b_w$, $w \in N(v)$ must have been modified since the last update of the satellite backups within C_v. This can be due to a deletion, creation, or update of a adjacent backup of $w.p$. Creation and deletion of backups within C_w only occurs finitely often, as prior to the second cycle C_w starts, they have been moved to the locations computed by the lower layer. An update of the adjacent backups of C_w requires a modification of $w.p$, which happens only finitely often. Hence, the number of cycles that C_v starts for reason (3) and during which $v.p$ is not modified is also finite.

It follows that each cell executed a finite number of cycles and thus a finite number of moves by Lemma 7.11. □

Theorem 7.21 (Slowdown)**.** *Let $e = \langle c_0, c_1, c_2, \ldots \rangle$ denote an execution of $\mathcal{A}_{FS}$ and let c_i denote the first configuration in which all cells are dialog-consistent and repaired and $v.k = k(v)$ for all $v \in V$. For $x \geq 0$, the primary configuration after $14 + 6x$ rounds of e is the result of an execution of at least x rounds of $\mathcal{A}$ starting in c_i.*

Proof. By Lemmas 7.10 and 7.16, all cells are dialog-consistent and repaired after 7 rounds and remain so thereafter. Hence, after round 7, no cell

is blocked or can become blocked. By Lemma 7.11, each cell is dialog-consistent, repaired, and unacknowledged at position PAUSED at least once within 7 more rounds. After the first round, $v.k = k(v)$ for all $v \in V$. Hence, c_i is reached within 14 rounds.

In c_i it holds $\neg \mathit{pair}(v)$ for all $v \in V$, since it is assumed that no minor components exist. Hence, by Lemma 7.17, all modifications of primary variables subsequent to c_i will be a result of moves of $\mathcal{A}$. Let v denote a node that is enabled with respect to $\mathcal{A}$ for the next 6 rounds. Without loss of generality, assume that $v.s = \text{EXECUTED}$. By Corollary 5.18, C_v finishes its current cycle and starts a new cycle, and advances to EXECUTED within 6 rounds. Note that C_v cannot become blocked as all cells are repaired. During the transition to EXECUTED, C_v executes a move of $\mathcal{A}$ for node v in line 16 of $\mathit{actionUpper}_Q(v)$ since either $\deg(v) > 1$ or $u.d_v \neq \text{SINGLE}$ for at least one $u \in N(v)$. Hence, within 6 rounds, any node that is continuously enabled with respect to $\mathcal{A}$ gets a chance to execute a move of $\mathcal{A}$. □

Proof of Theorem 7.7. Convergence follows from Theorems 7.18 and 7.20. Closure is shown in Theorem 7.19. Thus the transformed protocol is silent and self-stabilizing.

Let $T_{\mathcal{A}_L}$ and $T_{\mathcal{A}}$ denote the termination time of the lower layer and $\mathcal{A}$ in rounds respectively. After $\mathcal{O}(1) + \max\{T_{\mathcal{A}_L}, 6T_{\mathcal{A}}\}$ rounds of $\mathcal{A}_{FS}$, the lower layer has terminated. Also $\mathcal{A}$ has terminated by Theorem 7.21. By Corollary 7.13 and Lemma 7.11, $\mathcal{O}(1)$ rounds are needed until all cells have executed one complete cycle, during which the adjacent backups are updated and moved to their final locations. Again, $\mathcal{O}(1)$ rounds are needed until all cells have executed one complete cycle, during which the satellite backups are updated and moved to their final locations. Then $\mathcal{A}_{FS}$ terminates. □

7.4.2 Proof of Fault-Containing Super-Stabilization

The following proofs consider an execution $e = \langle c_0, c_1, c_2, \ldots \rangle$ that starts in a $(\Lambda, 1)$-faulty (E, c_0), where Λ denotes the class of topology changes which describe the addition and removal of a single edge. Furthermore, let $E_{\mathcal{L}}$ denote a topology, $c_{\mathcal{L}}$ a configuration, and v_f a node such that $c_{\mathcal{L}}$ is legitimate with respect to $E_{\mathcal{L}}$ and either E differs from $E_{\mathcal{L}}$ by a topology change of class Λ, c_0 is derived from $c_{\mathcal{L}}$ by perturbing variables of node v_f only, or both. To avoid an unnecessary complication of the proofs, minor components in E are not considered. They are reset to a legitimate configuration within a constant number of rounds as described

in Section 7.3.7.2. Note that the case where a minor component exists in $E_{\mathcal{L}}$ is considered in this section.

If the state corruption affected a minor component of two nodes $\{v, u\}$ in $E_{\mathcal{L}}$ and

$$v.b_u \neq \bot \wedge u.b_v \neq \bot \wedge$$
$$\neg \mathit{pairEnabled}_{\mathcal{A}}(v : v.p, u : v.b_u) \wedge \neg \mathit{pairEnabled}_{\mathcal{A}}(u : u.p, v : u.b_v)$$

is satisfied in c_0, then we define that $v_f = v$ (resp. $v_f = u$) if $v.id > u.id$ (resp. $u.id > v.id$). Accordingly, $c_{\mathcal{L}}$ is defined to be the configuration obtained from c_0 by setting $v.p := u.b_v$ (resp. $u.p := v.b_u$). This corresponds to the repair technique implemented by *pairReset*, which prefers to update the primary variable of the node with the larger identifier. If the state corruption affected a minor component in $E_{\mathcal{L}}$ of a single node and $\mathit{singleEnabled}_{\mathcal{A}}(v_f : v_f.p)$ is true in c_0, then the value of $v_f.p$ in $c_{\mathcal{L}}$ is assumed to be the one that is computed by $\mathit{actionUpper}_Q(v_f)$ in line 10.

It will be shown that $v_f.p$ is restored to the value in $c_{\mathcal{L}}$ and that none of the cells neighboring C_{v_f} execute moves of $\mathcal{A}$ before that happens. Hence, the corruption of $v_f.p$ cannot contaminate any other primary variables in the system, even if cells C_w, $w \notin N[v_f]$ start executing moves of $\mathcal{A}$ before $v_f.p$ has been restored to its value in $c_{\mathcal{L}}$.

For cell C_{v_f}, none of the backups have been corrupted. For any cell C_u with $u \in N(v_f)$, at most one backup was corrupted or deleted, or a forged backup $v_f.b_u$ was added by the state corruption. Also, dialog-variables may have been corrupted and responding instances on node v_f may have been deleted. For all other cells C_w, $w \notin N[v_f]$, neither the primary variable nor any of the (non-satellite) backups were corrupted.

Lemma 7.22. *Each cell C_u, $u \in V$ becomes dialog-consistent and unacknowledged with $u.s =$ PAUSED within at most two rounds. The primary variable $u.p$ is not modified. Neither adjacent nor satellite backups within C_u are modified, created, or deleted.*

Proof. Recall that in $c_{\mathcal{L}}$, the cell C_u is dialog-paused and $R(u) = N(u)$.

First, consider the case where C_u is initially non-operational in c_0. If the value of $(u.s, u.q)$ is not equal to (PAUSED, PAUSED), then (u, Q) resets $u.s$ and $u.q$ to PAUSED via Rule Q1 within the first round. Then it holds $u.s =$ PAUSED after the first round and during the the second round at least one responding instance (w, R_u), $w \in N(u)$ is spawned. $w.r_u$ is initialized with PAUSED. That renders C_u dialog-consistent. If $(u.s, u.q)$ is equal to (PAUSED, REPAIRED) in c_0, responding instances may be spawned in the first

round before (u, Q) performs the reset of $u.s$ and $u.q$. Then, (u, Q) does not perform the reset and C_u is dialog-consistent after the first responding instance is spawned.

The following cases assume that C_u is operational in c_0.

Case a) $v_f \notin R(u) \wedge v_f \neq u$. C_u is dialog-paused in c_0.

Case b) $v_f \in R(u)$. Since C_u was dialog-paused in $c_{\mathcal{L}}$, it holds $u.s = u.q =$ PAUSED and $w.r_u =$ PAUSED for all $v_f \neq w \in R(u)$ in c_0. If $v_f.r_u \neq$ PAUSED, then (v_f, R_u) resets $v_f.r_u$ to PAUSED within one round by Rule R1.

Case c) $v_f = u \wedge \neg \mathit{validQuery}(u)$. If $u.s = u.q =$ PAUSED then the claim holds. Otherwise, among the instances of C_u, only the center instance is enabled. It executes Rule Q1 within one round and resets $u.s$ and $u.q$ to PAUSED.

Case d) $v_f = u \wedge \mathit{validQuery}(u) \wedge u.q =$ PAUSED. It holds $u.s =$ EXECUTED and all response variables $v.r_u$ are equal to PAUSED. Hence the query by (u, Q) is already acknowledged and (v, Q) sets $u.s :=$ PAUSED by Rule Q3 within one round. Since $u.q =$ PAUSED, $\mathit{actionUpper}_Q(u)$ does not modify $u.p$ when invoked by Rule Q3.

Case e) $v_f = u \wedge \mathit{validQuery}(u) \wedge u.q \neq$ PAUSED. If $(u.s, u.q) =$ (PAUSED, REPAIRED) the claim holds. Otherwise $u.s \neq$ PAUSED and in C_u, only the center instance is enabled. Within one round, it resets $u.s$ and $u.q$ to PAUSED by Rule Q1.

In all cases, none of the moves modify a primary variable or delete, modify, or create any backups, including the satellite backups. □

Lemma 7.23. *Neither adjacent nor satellite backups within cell C_v are modified, created, or deleted as long as a non-repaired cell C_u, $u \in N(v)$ exists.*

Proof. By Lemma 7.22, C_v becomes dialog-consistent and unacknowledged with $v.s =$ PAUSED. No backups are modified, created or deleted. If C_v does not start a cycle, the claim holds. If C_v starts a cycle, the definition of $\mathit{dialogAcknowledged}(v)$ ensures that it holds $R(v) = N(v)$ before C_v reaches position REPAIRED. C_v is blocked during the transition REPAIRED $\rightarrow$ EXECUTED as long as any C_u, $u \in N(v)$ is not repaired. Hence, C_v does not attempt the transition EXECUTED $\rightarrow$ PAUSED during which the backups of $v.p$ and any satellite backups stored within C_v are modified. □

Lemma 7.24. *The primary variable $u.p$ of a cell C_u with $u \in N(v_f)$ is not modified as long as C_{v_f} is not repaired.*

Proof. By Lemma 7.22, C_u becomes dialog-consistent and unacknowledged with $u.s = \text{PAUSED}$. The primary variable $u.p$ has not been modified. If C_u does not start a cycle, the claim holds. If C_u starts a cycle, the definition of $\mathit{dialogAcknowledged}(u)$ ensures that it holds $R(u) = N(u)$ before C_u reaches position REPAIRED. In particular, every responding instance (w, R_u), $w \in N(u)$ executes Rule R2 in order to acknowledge the transition to REPAIRED. Since u is assumed not to be part of a minor component in E, it either holds $\deg(u) > 1$ or $w.d_u \neq \text{SINGLE}$ for any $w \in N(u)$.

Case a) $N(u) = \emptyset$ in $E_{\mathcal{L}}$. The forged backup $v_f.b_u$ may exist. No other backup of $u.p$ exists. If $v_f.b_u$ is a confirmation, then $\mathit{actionUpper}_Q(u)$ preserves the value of $u.p$. Otherwise, C_u is not repaired. Since v_f is not part of a minor component, another neighbor $u \neq w \in N(v_f)$ with $w.b_{v_f} \neq \perp$ exists. By Lemma 7.23, C_{v_f} did not yet create or delete any backups. Hence, $\mathit{formerPeer}(v_f) \neq u$ holds and $v_f.d_u$ is not set to MINOR. A second forged backup would be needed to obtain $v_f.d_u = \text{UPDATE}$. Hence $u.p$ is not modified by $\mathit{actionUpper}_Q(u)$ during the transition to REPAIRED since $\neg \mathit{singleEnabled}_{\mathcal{A}}(u : u.p)$.

Case b) u and v_f form a minor component in $E_{\mathcal{L}}$, $\deg(u) > 1$ in E, and $v_f.b_u = \perp$ in c_0. C_u is not repaired in c_0 and does not become repaired before reaching position REPAIRED. By Lemma 7.23, no cell is able to create, delete, or modify backups on node u. Thus $u.b_{v_f}$ remains the only backup that node u holds. This and $\mathit{actualBP}(u) = \emptyset$ yields that $\mathit{formerPeer}(u)$ returns v_f. Furthermore, $v_f.d_u$ is set to SINGLE. Hence $\mathit{actionUpper}_Q(u)$ calls $\mathit{pairReset}(u, v_f)$. Since $u.b_{v_f} \neq \perp$ and $\neg \mathit{pairEnabled}_{\mathcal{A}}(u : u.p, v_f : u.b_{v_f})$, the value of $u.p$ is not changed.

Case c) u and v_f form a minor component in $E_{\mathcal{L}}$, $\deg(u) > 1$ in E, and $v_f.b_u \neq \perp$ in c_0. If $v_f.b_u$ has not been corrupted, then it is a confirmation and thus $u.p$ is not modified by $\mathit{actionUpper}_Q(u)$ during the transition to REPAIRED. Otherwise $v_f.b_u$ is the only backup of $u.p$ and thus $\mathit{formerPeer}(u)$ returns v_f. Furthermore, $v_f.d_u$ is set to SINGLE. If $\neg \mathit{pairEnabled}_{\mathcal{A}}(u : v_f.b_u, v_f : v_f.p)$, then by definition of v_f, C_u does not finish the transition to REPAIRED before C_{v_f} is repaired and thus $\mathit{pairReset}(u, v_f)$ is never invoked. Otherwise, $\mathit{pairReset}(u, v_f)$ is called by $\mathit{actionUpper}_Q(u)$. Because the predicates $\mathit{pairEnabled}_{\mathcal{A}}(u : v_f.b_u, v_f : v_f.p)$ and $\neg \mathit{pairEnabled}_{\mathcal{A}}(u : u.p, v_f : u.b_{v_f})$ are satisfied, $u.p$ is not updated by $\mathit{pairReset}(u, v_f)$.

Case d) u and v_f form a minor component in $E_{\mathcal{L}}$ and $\deg(v_f) > 1$ in E. If $v_f.b_u = u.p$ in c_0, then $v_f.b_u$ is a confirmation and $u.p$ is not modified by $\mathit{actionUpper}_Q(u)$. Otherwise, if $v_f.b_u \neq u.p$, then C_u is not repaired in c_0 and doesn't become repaired before reaching REPAIRED. By

Lemma 7.23, C_{v_f} does not delete the backup $u.b_{v_f}$ or create additional backups on other responding instances of C_{v_f} as long as C_u is not repaired. Hence $\mathit{formerPeer}(v_f) = u$ and $v_f.d_u$ is set to MINOR. If $\neg \mathit{pairEnabled}_{\mathcal{A}}(u : v_f.b_u, v_f : v_f.p)$, then by definition of v_f, C_u does not finish the transition to REPAIRED before C_{v_f} is repaired and thus $\mathit{pairReset}(u, v_f)$ is never invoked. Otherwise, $\mathit{actionUpper}_Q(u)$ calls $\mathit{pairReset}(u, v_f)$. Because $\mathit{pairEnabled}_{\mathcal{A}}(u : v_f.b_u, v_f : v_f.p)$ and $\neg \mathit{pairEnabled}_{\mathcal{A}}(u : u.p, v_f : u.b_{v_f})$ hold, $u.p$ is not updated by $\mathit{pairReset}(u, v_f)$.

Case e) u and $w \neq v_f$ form a minor component in $E_{\mathcal{L}}$. Since u is assumed not to be part of a minor component in E it holds that $w \in N(u)$ in E. Since $w \neq v_f$, a confirming backup $w.b_u = u.p$ exists. Hence $u.p$ is not modified by $\mathit{actionUpper}_Q(u)$ during the transition to REPAIRED.

Case f) u is not part of a minor component in $E_{\mathcal{L}}$. As discussed at the beginning of Section 7.3, at least 2 backups are within the two-hop neighborhood of u. If a backup other than $v_f.b_u$ exists or if $v_f.b_u$ was not corrupted, then a confirmation of $u.p$ exists. Otherwise, a satellite backup within cell C_{v_f} must still exist by Lemma 7.23 and $v_f.d_u$ is set to KEEP. Hence $u.p$ is not modified by $\mathit{actionUpper}_Q(u)$ during the transition to REPAIRED.

C_u is blocked during the transition REPAIRED $\rightarrow$ EXECUTED as long as C_{v_f} is not repaired. Hence, C_u does not attempt the transition from EXECUTED to PAUSED during which $u.p$ may be modified. □

Lemma 7.25. *For any cell C_u with $u \notin N[v_f]$ it holds that any modification of $u.p$ is a result of moves of $\mathcal{A}$ executed during the transition* REPAIRED $\rightarrow$ EXECUTED.

Proof. By Lemma 7.22, C_u becomes dialog-consistent and unacknowledged with $u.s =$ PAUSED. The primary variable $u.p$ is not modified. If C_u starts a cycle, the definition of $\mathit{dialogAcknowledged}(u)$ ensures that $R(u) = N(u)$ before C_u reaches position REPAIRED. In particular, every responding instance (w, R_u) with $w \in N(u)$ executes Rule R2 in order to acknowledge the transition to REPAIRED. Since u is assumed not to be part of a minor component in E, either $\deg(u) > 1$ or $w.d_u \neq$ SINGLE for all $w \in N(u)$.

Case a) $N(u) = \emptyset$ in $E_{\mathcal{L}}$. Since u is not assumed to be part of a minor component in E, a neighbor $w \in N(u)$ with $\deg(w) > 1$ exists in E. Then $w.b_u = \perp$ and thus C_u is not repaired. If $w \in N(v_f)$, a forged satellite backup of $u.p$ may exist. If this backup is a confirmation, then $w.d_v$ is set to KEEP. Also, $w.d_v$ is not set to UPDATE, since a second forged backup does not exist. By Lemma 7.23, C_w cannot create the backup $u.b_w$. Hence

$\mathit{formerPeer}(w) \neq u$ and $w.d_v$ is not set to MINOR. Hence $u.p$ is not modified by $\mathit{actionUpper}_Q(u)$ during the transition to REPAIRED.

Case b) $N(u) \neq \emptyset$ in $E_{\mathcal{L}}$. Since u is not assumed to be part of a minor component in E, a neighbor $w \in N(u)$ with $w.b_u \neq \bot$ exists in E. Since $w \neq v_f$, $w.b_u$ must be a confirmation. Hence $u.p$ is not modified by $\mathit{actionUpper}_Q(u)$ during the transition to REPAIRED. □

Lemma 7.26. *The move that renders C_{v_f} repaired happens within at most 5 rounds and resets $v_f.p$ to its value in $c_{\mathcal{L}}$. It is the first modification of $v_f.p$. Any subsequent modification of $v_f.p$ is a result of an execution of moves of $\mathcal{A}$ executed during the transition* REPAIRED → EXECUTED.

Proof. By Lemma 7.22, C_{v_f} becomes dialog-consistent and unacknowledged with $v_f.s =$ PAUSED within two rounds. The primary variable $v_f.p$ has not been modified. Within one more round, missing responding instances are created. If C_{v_f} is already repaired, the claim holds by Lemma 7.17. From v_f not being repaired it follows that $\mathit{startUpper}_Q(v_f)$ is true and within one more round, (v_f, Q) starts a new cycle by setting $v_f.q :=$ EXECUTED. The definition of $\mathit{dialogAcknowledged}(v_f)$ ensures that $R(v_f) = N(v_f)$ holds before C_{v_f} reaches position REPAIRED. By the end of the fourth round, all responding instances have acknowledged the query for the transition to EXECUTED by executing Rule R2. Within at most four rounds, (v_f, Q) executes Rule Q3 and completes the transition to REPAIRED which renders C_{v_f} repaired. Since v_f is assumed not to be part of a minor component in E, either $\deg(v_f) > 1$ or $u.d_{v_f} \neq$ SINGLE for all $u \in N(v_f)$.

Case a) $N(v_f) = \emptyset$ in $E_{\mathcal{L}}$. No backup of $v_f.p$ can exist. Also, $u.d_{v_f} \neq$ KEEP and $u.d_{v_f} \neq$ UPDATE for all $u \in N(v_f)$ since no backup of $v_f.p$ can exist. Hence, $\mathit{actionUpper}_Q(v_f)$ computes a new value for $v_f.p$ such that $\neg \mathit{singleEnabled}_{\mathcal{A}}(v_f : v_f.p)$, which is assumed to be equal to $v_f.p$ in $c_{\mathcal{L}}$.

Case b) v_f and u form a minor component in $E_{\mathcal{L}}$ and $\deg(v_f) > 1$ in E. Note that $u.d_{v_f} =$ SINGLE. From C_{v_f} not being repaired $u.b_{v_f} \neq v_f.p$ follows. $u.b_{v_f}$ is the only backup of $v_f.p$, hence $\mathit{formerPeer}(v_f) = u$. Thus $\mathit{actionUpper}_Q(v_f)$ calls $\mathit{pairReset}(v_f, u)$ which sets $v_f.p := u.b_{v_f}$.

Case c) v_f and u form a minor component in $E_{\mathcal{L}}$ and $\deg(v_f) = 1$ in E. From C_{v_f} not being repaired $u.b_{v_f} \neq v_f.p$ follows. By Lemma 7.23, C_u cannot create, modify, or delete any backups of $u.p$ before C_{v_f} is repaired. If $v_f.b_u \neq \bot$, then it is the only existing backup of $u.p$ and $\mathit{formerPeer}(u) = v_f$ and thus $u.d_{v_f}$ is set to MINOR. Otherwise, if $v_f.b_u = \bot$, then $\mathit{actualBP}(u) = \emptyset$ and C_u must reach position REPAIRED to become repaired. If C_u is not repaired, $\mathit{formerPeer}(u) = v_f$ since $u.b_{v_f}$ is the only

backup node u holds in c_0 and no cell neighboring C_u can create another backup on node u by Lemma 7.23. If C_u is repaired, the value of $u.fp$ reflects the value of $formerPeer(u)$ prior to C_u becoming repaired. Hence, $u.d_{v_f}$ is set to MINOR. Thus in both cases $actionUpper_Q(v_f)$ calls $pairReset(v_f, u)$ and sets $v_f.p := u.b_{v_f}$.

Case d) v_f is not part of a minor component in $E_\mathcal{L}$. By the discussion at the beginning of Section 7.3, at least 2 backups exist within the 2-hop neighborhood of v_f. Let u denote a neighbor of v_f that holds a backup of $v_f.p$. If $u.b_{v_f}$ is a confirmation, then $actionUpper_Q(v_f)$ does not change the value of $v_f.p$. Otherwise, if $u.b_{v_f} \neq v_f.p$, then either a second backup $w.b_{v_f}$ with $w.b_{v_f} = u.b_{v_f}$ and $u \neq w \in N(v_f)$ or a satellite backup $w.sb_{u,v_f} = u.b_{v_f}$, $w \in N(u)$ exists. In the latter case, $u.d_v$ is set to UPDATE. In any case, $actionUpper_Q(v_f)$ updates $v_f.p$ with the value of $u.b_{v_f}$.

If C_{v_f} completes its current cycle it may execute a move of $\mathcal{A}$ during the transition to EXECUTED. Before C_{v_f} reaches position PAUSED none of the calls to $actionUpper_Q(v)$ modify $v_f.p$. By Lemma 7.16 cell C_{v_f} remains repaired and by Lemma 7.17 any modification of $v_f.p$ during a subsequent cycle of C_{v_f} is the result of a moves of $\mathcal{A}$. □

Lemma 7.27. *After C_{v_f} is repaired, it holds for all cells C_u, $u \in N(v_f)$ that any modification of $u.p$ is a result of an execution of moves of $\mathcal{A}$ executed during the transition* REPAIRED → EXECUTED.

Proof. The proof that C_u does not modify $u.p$ during the transition to REPAIRED is identical to the proof of Lemma 7.24, except for the following cases:

Case c) u and v_f form a minor component in $E_\mathcal{L}$, $\deg(u) > 1$ in E, and $v_f.b_u \neq \perp$ in c_0. If $v_f.b_u$ has not been corrupted, then it is a confirmation and thus $u.p$ is not modified by $actionUpper_Q(u)$ during the transition to REPAIRED. Otherwise C_u is not repaired and doesn't become repaired before reaching REPAIRED. $v_f.b_u$ is the only backup of $u.p$ and thus $formerPeer(u)$ returns v_f. Furthermore, $v_f.d_u$ is set to SINGLE and $actionUpper_Q(u)$ calls $pairReset(u, v_f)$. By Lemma 7.26, $v_f.p$ has been restored to its value in $c_\mathcal{L}$. Since C_u is not repaired, C_{v_f} becomes blocked afterwards and cannot modify $v_f.p$. Thus $\neg pairEnabled_\mathcal{A}(u : u.p, v_f : v_f.p)$ holds and $u.p$ is not modified by $pairReset(u, v_f)$.

Case d) u and v_f form a minor component in $E_\mathcal{L}$ and $\deg(v_f) > 1$ in E. If $v_f.b_u = u.p$ in c_0, then $v_f.b_u$ is a confirmation and $u.p$ is not modified by $actionUpper_Q(u)$. If $v_f.b_u \neq u.p$, then C_u is not repaired in c_0 and doesn't become repaired before reaching REPAIRED. By Lemma 7.23,

C_{v_f} does not delete the backup $u.b_{v_f}$ or create additional backups on other responding instances of C_{v_f} as long as C_u is not repaired. Hence $\mathit{formerPeer}(v_f) = u$ and $v_f.d_u$ is set to MINOR and $\mathit{actionUpper}_Q(u)$ calls $\mathit{pairReset}(u, v_f)$. By Lemma 7.26, $v_f.p$ has been restored to its value in $c_{\mathcal{L}}$. Since C_u is not repaired, C_{v_f} becomes blocked afterwards and cannot modify $v_f.p$. Thus $\neg \mathit{pairEnabled}_{\mathcal{A}}(u : u.p, v_f : v_f.p)$ holds and $u.p$ is not modified by $\mathit{pairReset}(u, v_f)$.

If C_u completes its current cycle it may execute a move of $\mathcal{A}$ during the transition to EXECUTED. Before C_u reaches position PAUSED, none of the calls to $\mathit{actionUpper}_Q(u)$ modify $u.p$. By reaching position REPAIRED, C_u has become repaired and remains repaired afterwards by Lemma 7.16. By Lemma 7.17 any modification of $u.p$ during a subsequent cycle of C_u is the result of moves of $\mathcal{A}$. □

Theorem 7.28 (Slowdown)**.** *For $x \geq 0$, the primary configuration after $7 + 6x$ rounds of an execution of $\mathcal{A}_{FS}$ starting in (E, c_0) is the result of an execution of at least x rounds of $\mathcal{A}$ starting in $(E, c_{\mathcal{L}})$.*

Proof. By Lemmas 7.24 to 7.26, a primary configuration is reached that is the result of an execution of $\mathcal{A}$ starting in the primary part of $c_{\mathcal{L}}$ within 5 rounds. By Lemmas 7.25 to 7.27, all subsequent modifications of a primary variable are the result of moves of $\mathcal{A}$. Note that within 2 more rounds (i.e., by the end of the 7th round), all cells are dialog-consistent and repaired and subsequently remain so by Lemmas 7.10 and 7.16.

Subsequently, every 6 rounds of the transformed protocol, one round of $\mathcal{A}$ is executed. This part of the proof is identical to the second half of the proof of Theorem 7.21. □

Theorem 7.29 (Containment Time)**.** *The containment time is constant.*

Proof. Recall that $\mathcal{A}$ is Λ-super-stabilizing and let $T_{\mathcal{A}} \in \mathcal{O}(1)$ denote the containment time of $\mathcal{A}$. After $T_{\mathcal{A}}$ rounds of any execution of $\mathcal{A}$ starting in the Λ-faulty $(E, c_{\mathcal{L}})$, all subsequent configurations satisfy $\mathcal{S}_{\mathcal{A}}$. By Theorem 7.21, after $7 + 6T_{\mathcal{A}}$ rounds of $\mathcal{A}_{FS}$ starting in (E, c_0), all subsequent configurations satisfy $\mathcal{S}_{\mathcal{A}}$. □

Theorem 7.30 (Fault-Gap)**.** *$\mathcal{A}_{FS}$ terminates after $\mathcal{O}(1) + \max\{T_{\mathcal{A}_L}, 6T_{\mathcal{A}}\}$ rounds, where $T_{\mathcal{A}_L}$ denotes the number of rounds $\mathcal{A}_L$ needs to terminate when starting in (E, c_0) and $T_{\mathcal{A}}$ denotes the number of rounds $\mathcal{A}$ needs to terminate starting in $(E, c_{\mathcal{L}})$.*

Proof. The lower layer and $\mathcal{A}$ terminate after $\max\{T_{\mathcal{A}_L}, 7+6T_{\mathcal{A}}\}$ rounds by Theorem 7.28. Subsequently, all cells update the adjacent backups for the last time and move them to their final locations within a constant number of rounds. After that, cells update the satellite backups for the last time and move to their final locations within a constant number of rounds. This holds by Corollary 7.13 and Lemma 7.11. □

Proof of Theorem 7.9. Follows from Theorems 7.8, 7.29 and 7.30. □

7.4.3 Proof of Fault-Containing Self-Stabilization

Let (E, c_0) and $(E_\mathcal{L}, c_\mathcal{L})$ be defined as in Section 7.4.2. However, assume that (E, c_0) is 1-faulty, i.e., only a state corruption affected node v_f but no topology change occurred and thus $E = E_\mathcal{L}$. By definition, any 1-faulty (E, c_0) is also $(\Lambda, 1)$-faulty and thus the proofs in Section 7.4.2 apply. The following proofs, consider an execution of $\mathcal{A}_{FS}$ that starts in the 1-faulty (E, c_0). Furthermore, let c_i denote the first configuration in which C_{v_f} is repaired and c_j the first configuration which satisfies $\mathcal{L}_\mathcal{A}$.

Lemma 7.31. *Before c_i is reached, $v_f.p$ is modified at most once and no other primary variable is modified.*

Proof. Before c_i, no cell C_u with $u \in N(v_f)$ modifies $u.p$ by Lemma 7.24. A node $w \notin N[v_f]$ is disabled with respect to $\mathcal{A}$ and thus does not change $w.p$ by Lemma 7.25. By Lemma 7.26, the move that yields c_i is the first move that modifies $v_f.p$. □

Lemma 7.32. *All nodes are disabled with respect to $\mathcal{A}$ in c_i and thus $j \leq i$. No primary variables are modified after c_i.*

Proof. Follows from Lemma 7.31 and the fact that $v_f.p$ is reset to the value in $c_\mathcal{L}$ by Lemma 7.26.

By Lemmas 7.25, 7.26, and 7.27, all modifications of a primary variable subsequent to c_i are results of moves of $\mathcal{A}$. Since $\mathcal{A}$ is disabled for all nodes in c_i, no modifications occur. □

Theorem 7.33 (Closure)**.** *All primary configurations subsequent to a legitimate primary configuration are also legitimate.*

Proof. By Lemmas 7.31 and 7.32, there are no modifications of primary variables other than the move that yields c_i which satisfies $\mathcal{L}_\mathcal{A}$. □

Theorem 7.34 (Containment time)**.** *The containment time is* 5 *rounds.*

Proof. By Lemma 7.26, c_i is reached within 5 rounds. The claims follows from Lemma 7.32. □

Theorem 7.35 (Contamination number)**.** *The contamination number is 1.*

Proof. By Lemma 7.32, $j \leq i$ and thus Lemma 7.31 implies that only $v_f.p$ is modified before c_j is reached. □

Theorem 7.36 (Fault-Gap)**.** *The transformed protocol terminates after* $\mathcal{O}(1) + T_L$ *rounds, where* T_L *denotes the number of rounds the lower layer needs to terminate starting at* (E, c_0)*.*

Proof. Follows from Theorem 7.30 for $E = E_{\mathcal{L}}$. Consider that $\mathcal{A}$ is disabled in c_i by Lemma 7.32. □

Proof of Theorem 7.8. Follows from Theorems 7.33 to 7.36. □

7.4.4 Serialization

The following ranking $r : \mathcal{I}_{\mathcal{A}_{FS}} \to \mathbb{N}$ is used to obtain partial serializations for steps of $\mathcal{A}_{FS}$ under the distributed scheduler. For notational convenience, the Boolean predicate $e_R(x)$ is used in the definition of $r(v, p)$. It is satisfied if and only if (v, p) is a responding instance and the x-th rule of (v, p) is enabled. Similarly, $e_Q(x)$ is true if and only if $p = Q$ and (v, Q) is enabled with respect to the x-th rule. If $e_R(x)$ is true, then u refers to the center instance (u, Q) that the responding (v, p) interacts with.

$$r(v,p) := \begin{cases} 1 & \text{if } e_R(2) \wedge u.q = \textsc{Repaired} \\ 2 & \text{if } e_R(2) \wedge u.q = \textsc{Paused} \wedge \mathit{updateSB}_{R_u}(v) \\ 3 & \text{if } e_R(2) \wedge u.q = \textsc{Paused} \wedge \neg\mathit{updateSB}_{R_u}(v) \\ 4 & \text{if } e_R(1) \vee (e_R(2) \wedge u.q = \textsc{Executed}) \\ 5 & \text{if } e_Q(2) \vee e_Q(3) \vee (e_Q(1) \wedge v.s = \textsc{Paused}) \\ 6 & \text{if } p = M \text{ and } (v, M) \text{ is enabled} \\ 7 & \text{if } e_Q(1) \wedge v.s \neq \textsc{Paused} \\ 8 & \text{if } p \in \mathcal{A}_L(v) \text{ and } (v, p) \text{ is enabled} \\ \bot & \text{otherwise} \end{cases}$$

For any step S of $\mathcal{A}_{FS}$ under the distributed scheduler, the ranking r is used to construct partial serializations of the form

$$\langle m_1, m_2, \ldots, m_k, S_{k+1}, S_{k+2}, S_{k+3}, S_{k+4} \rangle$$

where the sequence $m_1, m_2, \ldots, m_k$ consists of all instances within S that have ranks 1 to 4, sorted in ascending order by their rank. The sets S_{k+1} to $S_{k+4} \subseteq S$ contain the instances in S with ranks 5 to 8 respectively.

Instances of rank 1 set the decision variables. Their value is determined by the evaluation of $G_{\mathcal{A}}$ and by looking at the value of backups. Hence it is important that the primary variables and the backups have not been changed yet. Instances of rank 2 create, delete, or update satellite backups but no adjacent backups. This is only done by instances of rank 3. Instances of rank 4 reset a response-variable since the corresponding cell is non-dialog-consistent or they acknowledge a transition to EXECUTED, for which $actionUpper_{R_v}$ and $actionMiddle_{R_v}$ do not perform any actions. Instances of Q have rank 5, with the exception of certain instances of Q that perform a reset of $v.s$ and $v.q$. They have rank 7. Instances of rank 6 spawn and delete responding instances. All instances of the lower layer have rank 8. Lemma 7.40 proves the correctness of the partial serialization. In the following proofs, the set of variables that differ between c and $(c : m)$ with $m \in \mathcal{I}_{\mathcal{A}_{FS}}$ are called the *changeset* of m at c.

Lemma 7.37. *If cell C_v is not dialog-consistent, then (v, Q) is invariant under any (u, R_v) with $u \in N(v)$.*

Proof. Let c be a configuration, in which both $m_2 = (v, Q)$ and $m_1 = (u, R_v)$ are enabled. Let c' denote the configuration $(c : m_1)$.

Rules Q2 and Q3 of m_2 require $dialogConsistent(v)$ and are thus disabled in c. Only Rule Q1 of m_2 is enabled in c and hence $c \vdash (v.s, v.q) \neq$ (PAUSED, PAUSED). In conclusion, Rule R1 of m_1 is disabled and Rule R2 must be enabled in c. Thus $c \vdash (validQuery(v) \wedge u.r_v = v.s)$. From this and $c \vdash \neg dialogConsistent(v)$ it follows that a neighbor $w \in N(v)$ with $c \vdash w.r_v \notin \{v.s, v.q\}$ exists. Hence $w \neq u$ and $c'|_w = c|_w$ which implies $c' \vdash \neg dialogConsistent(v)$ and $c' \vdash r(m_2) = c \vdash r(m_2)$. Rule Q1 of m_2 sets both $v.s$ and $v.q$ to PAUSED. Hence $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$. □

Lemma 7.38. *If cell C_v is dialog-consistent, then (v, Q) is invariant under any (u, R_v) with $u \in N(v)$.*

Proof. It is shown that no configuration exists in which both (v, Q) and (u, R_v) are enabled. If $dialogConsistent(v)$, then Rule Q1 of (v, Q) and

Rule R1 of (u, R_v) are disabled. If Rule Q2 of (u, R_v) is enabled, then $\mathit{validQuery}(v)$ and $u.r_v = v.s$ holds. Hence, neither $\mathit{dialogPaused}(v)$ nor $\mathit{dialogAcknowledged}(v)$ and Rules Q2 and Q3 of (v, Q) are disabled. □

Lemma 7.39. *Let C_v and C_w denote two cells with $w \neq v$. Then (v, Q) is invariant under any (u, R_w), $u \in N(w)$.*

Proof. By Observation 4.25, only the case $u \in N[v]$ needs to be considered. At any configuration, a move (u, R_w) may change the response variables, the decision variables, or any of the backups of (u, R_w). Rank, guards and statements of (v, Q) only depend on variables from cell C_v and primary variables of neighboring cells, which are not in the changeset of (u, R_w). The only exception is $\mathit{startReset}_Q(v)$, which is part of the guard of Rule Q2.

Assume that $\mathit{startReset}_Q(v)$ is satisfied. This implies $\mathit{pairEnabled}_{\mathcal{A}}(v : v.p, u : u.p)$ and $v.p \neq f(v, u)$. If $w \in N(v)$ and $u = v$, then a move of (u, R_w) can change the value of $v.b_u$. It is either set to the value of $u.p$ or to the value $\bot$. In the former case, $\mathit{pairEnabled}_{\mathcal{A}}(v : v.p, u : u.p)$ implies $\mathit{pairEnabled}_{\mathcal{A}}(v : v.p, u : v.b_u)$ after the move of (u, R_w) since $u.p = v.b_u$. Hence in both cases, $(v.b_u = \bot \vee \mathit{pairEnabled}_{\mathcal{A}}(v : v.p, u : v.b_u)$ and thus $\mathit{startReset}_Q(v)$ remains satisfied after the move of (u, R_w). □

Lemma 7.40. *The ranking r is a weak invariancy ranking. For instances of rank less than 5, r is a proper invariancy-ranking.*

Proof. Let m_2 and m_1 with $m_2 \neq m_1$ denote two instances of nodes v_2 and v_1 and cells u_2 and u_1 respectively. For the given ranks r_2, r_1, let c denote a configuration which satisfies $r_2 = c \vdash r(m_2)$ and $r_1 = c \vdash r(m_1)$ and c' denote the configuration $(c : m_1)$. By Observation 4.25, only the case where m_2 and m_1 are neighboring instances is considered.

First, it is shown that r is a proper invariancy ranking for instances of rank less than 5:

Case a) $r_2 = 1 \wedge r_1 = 1$: The assumption yields Rule R2 of both m_2 and m_1 is enabled in c and that $c \vdash u_1.q = u_2.q = \textsc{Repaired}$. Since $c \vdash u_1.q = \textsc{Repaired}$, the changeset of m_1 at c is $\{v_1.r_{u_1}, v_1.d_{u_1}\}$. Hence, $c' \vdash u_2.q = \textsc{Repaired}$ holds and thus the guard of Rule R2 of m_2 only depends on the variables $v_2.r_{u_2}$, $u_2.s$, $u_2.q$, and backups within cell u_2 which are not in the changeset of m_1. Thus Rule R2 of m_2 remains enabled in c' and thus $c' \vdash r(m_2) = c \vdash r(m_2)$. Since $c' \vdash u_2.q = \textsc{Repaired}$, the output of $(c' : m_2)|_{m_2}$ only depends on the variable $u_2.q$, $v_2.fp$, primary variables, and backups which are not part of the changeset of m_1.

Case b) $r_2 = 2 \wedge r_1 \in \{1, 2\}$: At worst, the changeset of m_1 includes $\{v_1.r_{u_1}, v_1.b_{u_1}, v_1.d_{u_1}\}$ and satellite backups. Hence Rule R2 of m_2 is still enabled in c'. It follows that $c' \vdash r(m_2) = c \vdash r(m_2)$. Since m_2 is of rank 2, it sets $v_2.r_{u_2}$ to $u_2.q$ and possibly updates several satellite backups with the values of adjacent backups.. Both $u_2.q$ and adjacent backups are not among the changeset of m_1 and thus $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$.

Case c) $r_2 = 3 \wedge r_1 \in \{1, 2, 3\}$: At worst, the changeset of m_1 includes $\{v_1.r_{u_1}, v_1.b_{u_1}, v_1.d_{u_1}\}$, satellite backups, and adjacent backups of $u_1.p$. Hence Rule R2 of m_2 is still enabled in c'. It follows that $c' \vdash r(m_2) = c \vdash r(m_2)$. Since m_2 is of rank 3, it sets $v_2.r_{u_2}$ to $u_2.q$ and possibly updates $v_2.b_{u_2}$ with the value of $u_2.p$. Both $u_2.q$ and $u_2.p$ are not among the changeset of m_1 and thus $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$.

Case d) $r_2 = 4 \wedge r_1 \in \{1, 2, 3, 4\}$: At worst, the changeset of m_1 includes $\{v_1.r_{u_1}, v_1.b_{u_1}, v_1.d_{u_1}\}$, satellite backups, and adjacent backups of $u_1.p$. The predicate *repaired*(v_2) is the only part of the guards of m_2 which might reference these variables, namely in the case $v_2 = u_1$. By Lemma 7.16 $c \vdash$ *repaired*$(v) \Rightarrow c' \vdash$ *repaired*(v). Hence the rule of m_2 that is enabled in c is still enabled in c'. It follows that $c' \vdash r(m_2) = c \vdash r(m_2)$. Since m_2 is of rank 4, it sets $v_2.r_{u_2}$ to $u_2.q$ but does not modify any backups or decision variables. Thus $(c' : m_2)|_{m_2} = (c : m_2)|_{m_2}$.

The above also shows that r is also a weak invariancy ranking for instances of rank less than 5. It remains to show that this also holds if instances of rank 5 or larger are involved. Let m_2 denote an instance of node v_2 and cell u_2 with $r_2 = c \vdash r(m_2)$ and c' denote the configuration $(c : S)$, where S is a set of instances $m \in \mathcal{I}_{\mathcal{A}_{FS}}$ with $r_1 = c \vdash r(m)$.

Case f) $r_2 = \in \{5, 7\} \wedge r_1 \in \{1, 2, 3, 4\}$: Lemma 7.39 applies if $u_1 \neq u_2$. Otherwise either Lemma 7.38 or Lemma 7.37 applies.

Case e) $r_2 = 6 \wedge r_1 \in \{1, 2, 3, 4, 5\}$: m_2 is an instance of M. The set of instances deleted or spawned by m_2 remains unchanged if any instances $(v, Q) \in S$ does not set $v.s$ to a value different from PAUSED. Rules Q1 and Q2 do not do so. Furthermore, (v, Q) must be disabled with respect to Rule Q3 by definition of *dialogAcknowledged*(v) if $R(v) \neq N(v)$.

Case f) $r_2 = 7 \wedge r_1 = 5$: Cell C_{u_2} is non-dialog-consistent in c. A move of an instance (v, Q) with $v \neq u_2$ cannot change this. Hence, m_2 remains enabled and set $v.s$ and $v.q$ to PAUSED.

Case g) $r_2 = 7 \wedge r_1 = 6$: Cell C_{u_2} is non-dialog-consistent with $u_2.s \neq$ PAUSED in c. Thus, no responding instance of cell C_{u_2} is spawned by any instance of M in S.

Case h) $r_2 = 8 \wedge r_1 \in \{1, 2, 3, 4, 5, 6, 7\}$: Since $\mathcal{A}_{FS}$ is a collateral composition, no instance of $\mathcal{A}_L$ reads variables of any instances with a rank smaller than 8. □

7.5 Optimizations

Several optimizations and modifications can be applied to the transformation given in Section 7.3. First of all, the alternative state-machine proposed in Section 5.8.6 for $\mathcal{A}_{FL}$ can also be used for $\mathcal{A}_{FS}$. It lowers the slowdown to 4 rounds and may also decrease the containment time of $\mathcal{A}_{FS}$. Furthermore, to save space, the center instance of any dialog-paused cell C_v may set the variables $v.bptr$ and $v.sbptr$ to the empty set.

As suggested in Section 6.7, choosing the composition $\mathcal{A}_L + \mathcal{A}$ as the input to $\mathcal{A}_{FS}$ has the advantage that the fault-containment implemented by $\mathcal{A}_{FS}$ would also repair corruptions of variables of the lower layer. Furthermore, the fault-gap in Theorem 7.8 would be lowered to $\mathcal{O}(1)$ rounds and the fault-gap in Theorem 7.9 would be lowered to $\mathcal{O}(1) + 6 \max\{T_{\mathcal{A}_L}, T_{\mathcal{A}}\}$ rounds, where $T_{\mathcal{A}_L}$ and $T_{\mathcal{A}}$ denote the termination time of the lower layer and $\mathcal{A}$ for Λ-faulty initial configurations in rounds. If the termination time of the lower layer for Λ-faulty initial configuration is significantly lower than the termination time for $(\Lambda, 1)$-faulty configurations, then using this enhancement is desirable. The disadvantage is that the stabilization of $\mathcal{A}_L$ is slowed down by a factor of 6.

7.6 Discussion

To the best of our knowledge, this chapter is the first work to implement the repair of state corruptions if a topology change occurs simultaneously. Pectu and Faltings published a self-stabilizing protocol for multiagent combinatorial optimization [PF05]. They describe two extensions: one to make the protocol fault-containing and one to make it super-stabilizing with respect to topology changes. However, it is unclear whether both extensions can be combined, and whether they remain functional if a state corruption happens in parallel with a topology change.

Besides super-stabilizing protocols, the transformation presented in this chapter can also be applied to other algorithms with special properties regarding topology changes. An example is the third leader election algorithm given in [DLP10]. It provides a high degree of leadership stability. This means that each node changes its choice of leader at most once and that it

re-elects a node which also was a leader prior to the topology change, provided that there is at least one former leader in the connected component. Furthermore, during recovery from a topology change, no node changes its choice of leader more than once. The topology changes can be of arbitrary scale. These properties cannot be expressed as any sort of safety predicate. If the transformation was applied to this algorithm, $\mathcal{A}_{FS}$ would first repair the state corruption and then execute moves of the leader election algorithm. Hence, the properties of the leader election algorithm are preserved even for $(\Lambda, 1)$-faulty configurations. Only the presence of minor components after or prior to the fault may result in cases where the properties are not preserved.

The subject of topology changes during stabilization is discussed by Derhab and Badache [DB08]. Their leader-election protocol is self-stabilizing even if topology changes occur frequently during stabilization. The given transformation also tolerates topology changes that happen during stabilization under certain conditions. Assume that cells satisfy the prerequisites of Lemma 7.17, such that all subsequent modifications of any primary variables are a result of moves of the input algorithm $\mathcal{A}$. For edge additions, a cell will continue with the current cycle, and spawn missing responding instances before the next cycle. No modification of the primary variables occurs that might impact the stabilization of $\mathcal{A}$. For edge removals it must hold that all affected cells are still repaired after the topology change. This ensures that no primary variable is modified in a way that is not the result of moves of $\mathcal{A}$. It takes cells at most one full cycle, i.e., a constant number of rounds, to adapt the placement of backups to the new topology. Then the transformed protocol is ready to tolerate another topology change.

Once a legitimate configuration has been reached, $\mathcal{A}_{FS}$ recovers from a state corruption within a constant number of rounds, even if a single edge has been added or removed simultaneously. However, we believe that this also holds for the crash and the recovery of a node. From the perspective of a single node, there is no difference between the crash of a neighbor and the removal of the corresponding edge. In particular, sufficiently many backups remain reachable unless the node is part of a minor component. Also the addition of multiple edges adjacent to the same node was taken into account during the design of the transformation. In particular the detection of former minor components remains functional. Furthermore, the transformation can handle the removal of more than one edge if more backups per node are created. Backups on all nodes within 2-hop distance of each node are sufficient to handle the removal of any number of edges. Topology changes that add and remove edges at the same time remain problematic,

in particular if the topology change disconnects a minor component from one graph component and connects it to different component.

The presented transformation can handle multiple state corruptions if the distance between any pair of affected nodes is at least 5 hops. These state corruptions can also be accompanied by multiple topology changes if the distance between any pair of edges that were removed or added is large enough. This stems from the fact that the techniques used to detect and repair the state corruption are based entirely on information in the local neighborhood of the individual nodes.

If a large number of communication links is expected to fail in a very small area, then increasing the number of adjacent and satellite backups per node can increase the probability that state corruptions are successfully repaired. In particular, if all neighbors and all nodes at distance 2 of each node store backups of the nodes primary variables, then an arbitrary number communication links may fail. Unless the node is part of a minor component, at least 2 backups remain accessible. Hence, there is a trade-off between space requirements and the degree of resilience against topological changes.

We are not aware of methods to further reduce the space overhead without increasing the time-complexity of the transformation. We conjecture that less than four backups do not suffice to guarantee the bounds given in Section 7.4.

For the design of $\mathcal{A}_{FS}$, we assumed locally unique identifiers. We did not investigate whether it is possible to use the transformation with weakly unique identifiers. In particular, the function f used by procedure *pairReset* would have to be adjusted. It is likely that a reset for minor components can be implemented if $\mathcal{A}$ is assumed to be self-stabilizing with weakly unique identifiers.

The definition of fault-containing super-stabilization does not include a definition of a contamination number. Intuitively, it should describe the number of nodes which are impaired by the state corruption. $\mathcal{A}_{FS}$ tightly contains the state corruption, i.e., no node executes moves of $\mathcal{A}$ that would read any corrupted primary variable. The problem with defining a contamination number is that nodes at distance larger than 1 of the state corruption may already start executing moves of $\mathcal{A}$ due to the topology change. Hence, simply counting the nodes changing their primary variables is not suitable. Informally, a possible definition of the contamination number would be the number of nodes that assign values to their primary variables that are the result of a computation which incorporated the values of corrupted primary variables.

8 Concluding Remarks

This thesis presents two new transformations for fault-containment, i.e., containing the effects of a single node's state corruption. The transformation $\mathcal{A}_{FL}$ described in Chapter 5 assumes a static topology and adds fault-containment to any silent self-stabilizing algorithm. Chapter 7 describes the transformation $\mathcal{A}_{FS}$ which is suitable for dynamic networks and adds fault-containment to silent super-stabilizing algorithms.

$\mathcal{A}_{FL}$ has constant containment time, contamination number, and slowdown factor. The constant fault-gap and the fault-impact radius of 2 are considerable improvements over previously known transformations for asynchronous systems. After a state corruption, only a constant number of rounds is required to prepare for the containment of another state corruption in the same region. This greatly increases the availability of systems where small-scale state corruptions are expected to occur frequently. This was achieved using a technique for local synchronization instead of global synchronization. The presented transformation is the first general technique for fault-containing self-stabilization in asynchronous systems that has a fault-gap independent of the number n of nodes and the maximal degree Δ of the topology. Furthermore, no a priori knowledge about n and Δ is required. Hence, the transformation is especially suitable for large systems.

Chapter 7 introduces the notion of fault-containing super-stabilization, which transfers the notion of fault-containment to the setting of dynamic networks. $\mathcal{A}_{FS}$ provides fault-containing super-stabilization and inherits the super-stabilization property from the input algorithm even in case a state corruption occurs simultaneously with the topology change. The transformation increases the containment time of the super-stabilizing input algorithm by a constant number of rounds only.

It is shown that two backups suffice for fault-containing self-stabilization and four backups suffice for fault-containing super-stabilization. In order to balance the loads of the nodes, the algorithm $\mathcal{A}_P$ is proposed in Chapter 6. $\mathcal{A}_P$ solves the local k-placement problem and computes backup placements suitable for $\mathcal{A}_{FL}$ with locally minimal variance in $\mathcal{O}(n\Delta)$ rounds. The bound $\mathcal{O}(n\Delta)$ on the stabilization time of the local k-placement algorithm may not

be tight. It is considered future work to either find an example for which the stabilization time $\mathcal{O}(n\Delta)$ is actually reached or to further improve the analysis.

$\mathcal{A}_P$ can be extended to compute a backup placement suitable for $\mathcal{A}_{FS}$. However, the fault-gap of $\mathcal{A}_{FS}$ depends on the termination time of the placement algorithm after a topology change. It may be possible to optimize the fault-gap of the transformation by using placement algorithms that combine slightly worse balanced loads with a smaller termination time after topology changes. This trade-off can motivate the development of further placement algorithms. The local k-placement problem may also have applications in other areas of distributed computing where redundancy is required. Also, metrics other than the variance (or equivalently the standard deviation) may be of interest.

The presented algorithms are designed for the unfair distributed scheduler and require weakly or locally unique identifiers. Via the serialization technique introduced in Chapter 4, it is possible to construct (partial) serializations for the given algorithms. This simplifies the reasoning in the proofs, as it reduces the degree of parallelism that has to be considered in the proofs.

The transformations do not provide fault-containment for non-silent algorithms. A core problem is that non-silent algorithms may continuously change the local states of the nodes and the backups frequently become inconsistent. State corruptions that happen during the stabilization, as opposed to state corruptions that happen only in a legitimate configuration, pose another problem that has not been tackled by the given transformations. We believe that these two problems are related. To show the feasibility and implement transformations that are able to provide this kind of fault-containment is considered future work.

The transformations only consider the corruption of a single node's state. State corruptions of larger scale are not contained by the transformations and may lead to contamination. This holds in particular if the state corruptions affect the states of many adjacent nodes, e.g., all instances of a cell are affected. Concepts like adaptive stabilization exist, but the space overhead of the transformation sketched in [KPS99] is tremendous.

It would also be desirable to provide fault-containment in spite of topology changes more far-reaching than the failure or recovery of a single link. By increasing the number of backups created by $\mathcal{A}_{FS}$, it can be tuned to tolerate the failure or recovery of a larger number of communication links. However, minor components are problematic. It remains an open problem, whether $\mathcal{A}_{FS}$ can be extended to successfully identify and reset minor com-

ponents that have been disconnected and reconnected to different nodes by the topology change. It is worth mentioning that $\mathcal{A}_{FS}$ partly tolerates topology changes that occur during stabilization and are not accompanied by state corruptions. Hence, $\mathcal{A}_{FS}$ may be suitable for systems where the rate of topology changes during stabilization can be controlled or is expected to be low.

Bibliography

[AG94] Anish Arora and Mohamed Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[AHO03] Ravindra K. Ahuja, Dorit S. Hochbaum, and James B. Orlin. Solving the convex cost integer dual network flow problem. *Management Science*, 49(7):950–964, 2003.

[Ang80] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the 12th annual ACM Symposium on Theory of Computing*, pages 82–93. ACM, 1980.

[APSV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 268–277. IEEE Computer Society, 1991.

[APSVD94] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset (extended abstract). In *Proceedings of the 8th International Workshop on Distributed Algorithms*, volume 857 of *Lecture Notes in Computer Science*, pages 326–339. Springer, 1994.

[AV91] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols (extended abstract). In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 258–267. IEEE Computer Society, 1991.

[BDGM02] Joffroy Beauquier, Ajoy K. Datta, Maria Gradinariu, and Frédéric Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *Chicago Journal of Theoretical Computer Science*, 2002(1), 2002.

[BDH06] Joffroy Beauquier, Sylvie Delaët, and Sammy Haddad. A 1-strong self-stabilizing transformer. In *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 4280 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2006.

[BGK99] Joffroy Beauquier, Christophe Genolini, and Shay Kutten. Optimal reactive k-stabilization: the case of mutual exclusion. In *Proceedings of the 18th annual ACM symposium on Principles of distributed computing*, pages 209–218. ACM, 1999.

[BGM89] James E. Burns, Mohamed G. Gouda, and Raymon E. Miller. On relaxing interleaving assumptions. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report STP-379-89*, Austin, Texas, USA, 1989.

[BHKPS06] Janna Burman, Ted Herman, Shay Kutten, and Boaz Patt-Shamir. Asynchronous and fully self-stabilizing time-adaptive majority consensus. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, volume 3974 of *Lecture Notes in Computer Science*, pages 146–160. Springer, 2006.

[BPBRT10] Lélia Blin, Maria Potop-Butucaru, Stephane Rovedakis, and Sébastien Tixeuil. Loop-free super-stabilizing spanning tree construction. In *Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 6366 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2010.

[CYH91] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39(3):147–151, 1991.

[DB08] Abdelouahid Derhab and Nadjib Badache. A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile networks. *IEEE Transactions on Parallel and Distributed Systems*, 19:926–939, 2008.

[DGX07] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Probabilistic fault-containment. In *Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 4838 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2007.

[DGX11] Anurag Dasgupta, Sukumar Ghosh, and Xin Xiao. Fault containment in weakly stabilizing systems. *Theoretical Computer Science*, 412(33):4297–4311, 2011.

[DH95] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, page 255. ACM, 1995.

[DH97] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997(4), 1997.

[DH99] Shlomi Dolev and Ted Herman. Parallel composition of stabilizing algorithms. In *Proceedings of the 1999 ICDCS Workshop on Self-stabilizing Systems*, pages 25–32. IEEE Computer Society, 1999.

[DHR+11] Stephane Devismes, Karel Heurtefeux, Yvan Rivierre, Ajoy K. Datta, and Lawrence L. Larmore. Self-stabilizing small k-dominating sets. In *Proceedings of the 2nd International Conference on Networking and Computing*, pages 30–39. IEEE Computer Society, 2011.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[Dij86] Edsger W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[DIM90] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proceedings of the 9th annual ACM symposium on Principles of distributedcomputing*, pages 103–117. ACM, 1990.

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[DLP10] Ajoy K. Datta, Lawrence Larmore, and Hema Piniganti. Self-stabilizing leader election in dynamic networks. In *Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 6366 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2010.

[DLV11] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science*, 412(40):5541–5561, 2011.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, USA, 2000.

[DTY08] Stéphane Devismes, Sébastien Tixeuil, and Masafumi Yamashita. Weak vs. self vs. probabilistic stabilization. In *Proceedings of the 28th International Conference on Distributed Computing Systems*, pages 681–688. IEEE Computer Society, 2008.

[FDB94] M. Flatebo, Ajoy K. Datta, and B. Bourgon. Self-stabilizing load balancing algorithms. In *Proc. IEEE 13th Annual Int. Phoenix Conf. on Computers and Communications*, page 303. IEEE Computer Society, April 1994.

[GCH06] Mohamed G. Gouda, Jorge A. Cobb, and Chin-Tser Huang. Fault masking in tri-redundant systems. In *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 4280 of *Lecture Notes in Computer Science*, pages 304–313. Springer, 2006.

[GG96] Sukumar Ghosh and Arobinda Gupta. An exercise in fault-containment: Self-stabilizing leader election. *Information Processing Letters*, 59(5):281–288, 1996.

[GGH+04] Martin Gairing, Wayne Goddard, Stephen T. Hedetniemi, Petter Kristiansen, and Alice A. McRae. Distance-two information in self-stabilizing algorithms. *Parallel Processing Letters*, 14(03n04):387–398, 2004.

[GGHP96] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54. ACM, 1996.

[GGHP07] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing distributed protocols. *Distributed Computing*, 20(1):53–73, 2007.

[GGP96] Sukumar Ghosh, Arobinda Gupta, and Sriram V. Pemmaraju. A fault-containing self-stabilizing spanning tree algorithm. *Journal of Computing and Information*, 2(1):322–338, 1996.

[GGP97a] Sukumar Ghosh, Arobinda Gupta, and Sriram V. Pemmaraju. Fault-containing network protocols. In *Proceedings of the 1997 ACM symposium on Applied computing*, pages 431–437. ACM, 1997.

[GGP97b] Sukumar Ghosh, Arobinda Gupta, and Sriram V. Pemmaraju. A self-stabilizing algorithm for the maximum flow problem. *Distributed Computing*, 10(4):167–180, 1997.

[GH91] M.G. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17:911–921, 1991.

[GH99] Sukumar Ghosh and Xin He. Scalable self-stabilization. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Workshop on Self-Stabilizing Systems*, pages 18–24. IEEE Computer Society, 1999.

[GH00] Sukumar Ghosh and Xin He. Fault-containing self-stabilization using priority scheduling. *Information Processing Letters*, 73(3-4):145–151, 2000.

[GHJT08] Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, and Vilmar Trevisan. Distance-k knowledge in self-stabilizing algorithms. *Theoretical Computer Science*, 399(1–2):118–127, 2008.

[GJR+10] Dominik Gall, Riko Jacob, Andrea Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. Time complexity of distributed topological self-stabilization: The case of graph linearization. In *Proceedings of the 9th Latin American Symposium on Theoretical Informatics*, volume 6034 of *Lecture Notes in Computer Science*, pages 294–305. Springer, 2010.

[GM91] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[Gou01] Mohamed G. Gouda. The theory of weak stabilization. In *Proceedings of the 5th Workshop on Self-Stabilizing Systems*, volume 2194 of *Lecture Notes in Computer Science*, pages 114–123. Springer, 2001.

[GP99] Felix C. Gärtner and Henning Pagnia. Self-stabilizing load distribution for replicated servers on a per-access basis. In *Proc. 19th IEEE Int. Conf. on Distributed Computing Systems, Workshop on Self-Stabilizing Systems*, pages 102–109. IEEE Computer Society, June 1999.

[Gri81] David Gries. *The science of programming.* Springer, 1981.

[GT07] Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems*, page 46. IEEE Computer Society, 2007.

[Gup97] Arobinda Gupta. *Fault-Containment in Self-Stabilizing Distributed Systems.* PhD thesis, University of Iowa, Iowa City, IA, USA, 1997.

[Gä99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[Gä01] Felix Christoph Gärtner. *Formale Grundlagen der Fehlertoleranz in verteilten Systemen.* PhD thesis, TU Darmstadt, July 2001.

[Gä03] Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical Report IC/2003/38, Swiss Federal Institute of Technology, 2003.

[Hau08] Bernd Hauck. Worst-case analysis of a self-stabilizing algorithm-computing a weakly connected minimal dominating set. Technical Report urn:nbn:de:gbv:830-tubdok-5126, Hamburg University of Technology, Hamburg, Germany, October 2008.

[HC92] Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2):109–117, 1992.

[Her90] Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[Her92] Ted Richard Herman. *Adaptivity through distributed convergence*. PhD thesis, University of Texas at Austin, Austin, TX, USA, 1992.

[Her00] Ted Herman. Superstabilizing mutual exclusion. *Distributed Computing*, 13:1–17, 2000.

[HHJS03] S. M. Hedetniemi, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Computers and Mathematics with Applications*, 46(5–6):805–811, 2003.

[HP00] Ted Herman and Sriram Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Information Processing Letters*, 73(1-2):41–46, 2000.

[Hua06] Tetz C. Huang. An efficient fault-containing self-stabilizing algorithm for the shortest path problem. *Distributed Computing*, 19(2):149–161, 2006.

[Kes88] Joep L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29(1):39–42, 1988.

[KM06] H. Kakugawa and T. Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, page 8. IEEE Computer Society, April 2006.

[KMW04] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally! In *Proceedings of the 23rd annual ACM symposium on Principles of distributed computing*, pages 300–309. ACM, 2004.

[KP93] Shmuel Katz and Kenneth Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[KP95] Shay Kutten and David Peleg. Fault-local distributed mending (extended abstract). In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, pages 20–27. ACM, 1995.

[KP99] Shay Kutten and David Peleg. Fault-local distributed mending. *Journal of Algorithms*, 30(1):144–165, 1999.

[KPS99] Shay Kutten and Boaz Patt-Shamir. Stabilizing time-adaptive protocols. *Theoretical Computer Science*, 220(1):93–111, 1999.

[KPS04] Shay Kutten and Boaz Patt-Shamir. Adaptive stabilization of reactive protocols. In *Proceedings of the 2004 IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 396–407. Springer, 2004.

[KR05] Bong-Jun Ko and Dan Rubenstein. Distributed self-stabilizing placement of replicated resources in emerging networks. *IEEE/ACM Trans. Networking*, 13(3):476–487, June 2005.

[KR06] David R. Karger and Matthias Ruhl. Simple efficient load-balancing algorithms for peer-to-peer systems. *Theory Comput. Syst.*, 39(6):787–804, 2006.

[KRR02] Jussi Kangasharju, James W. Roberts, and Keith W. Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25(4):376–383, 2002.

[KUFM02] Yoshiaki Katayama, Eiichiro Ueda, Hideo Fujiwara, and Toshimitsu Masuzawa. A latency optimal superstabilizing mutual exclusion protocol in unidirectional rings. *Journal of Parallel and Distributed Computing*, 62(5):865–884, 2002.

[Lam84] Leslie Lamport. Solved problems, unsolved problems and nonproblems in concurrency. In *Proceedings of the 3rd annual ACM symposium on Principles of distributed computing*, pages 1–11. ACM, 1984.

[LG91] X. Lin and S. Ghosh. Self-stabilizing maxima finding. In *Proceedings of the 29th Allerton Conference*, 1991.

[LH03] Ji-Cherng Lin and Tetz C. Huang. An efficient fault-containing self-stabilizing algorithm for finding a maximal independent set. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):742–754, 2003.

[LW11] Christoph Lenzen and Roger Wattenhofer. Tight bounds for parallel randomized load balancing: extended abstract. In *Proc. 43rd ACM Symp. on Theory of Computing*, pages 11–20. ACM, 2011.

[MM07] Fredrik Manne and Morten Mjelde. A self-stabilizing weighted matching algorithm. In *Stabilization, Safety, and Security of Distributed Systems*, volume 4838 of *Lecture Notes in Computer Science*, pages 383–393. Springer, 2007.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[NS93] Moni Naor and Larry Stockmeyer. What can be computed locally? In *Proceedings of the 25th annual ACM symposium on Theory of computing*, pages 184–193. ACM, 1993.

[PD02] Vinayaka Pandit and Dhananjay M. Dhamdhere. Self-stabilizing maxima finding on general graphs. Technical Report RI02009, IBM T.J. Watson Research Center, April 2002.

[Pel00] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

[PF05] Adrian Petcu and Boi Faltings. Superstabilizing, fault-containing distributed combinatorial optimization. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*, pages 449–454. AAAI Press, 2005.

[POD82] *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 1982.

[SBKF07] Sabina Serbu, Silvia Bianchi, Peter Kropf, and Pascal Felber. Dynamic load sharing in peer-to-peer systems: When some peers are more equal than others. *IEEE Internet Computing*, 11:53–61, 2007.

[SRR95] Sandeep K. Shukla, Daniel J. Rosenkrantz, and S. S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks (extended abstract). In *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, pages 7.1–7.15, 1995.

[SS12] Thomas Sauerwald and He Sun. Tight bounds for randomized load balancing on arbitrary network topologies. *CoRR*, abs/1201.2715, 2012.

[SX07] Pradip K. Srimani and Zhenyu Xu. Self-stabilizing algorithms of constructing spanning tree and weakly connected minimal dominating set. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, page 3, 2007.

[Tel00] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000.

[TH11] Volker Turau and Bernd Hauck. A new analysis of a self-stabilizing maximum weight matching algorithm with approximation ratio 2. *Theoretical Computer Science*, 412(40):5527–5540, 2011.

[The00] Oliver Theel. Exploitation of ljapunov theory for verifying self-stabilizing algorithms. In *Proceedings of the 14th International Symposium on Distributed Computing*, volume 1914 of *Lecture Notes in Computer Science*, pages 213–251. Springer, 2000.

[Tix09] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook*, volume 2, chapter 26: Self-Stabilizing Algorithms. CRC Press, Taylor & Francis Group, 2nd edition, 2009.

[Tur07] Volker Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Information Processing Letters*, 103(3):88–93, 2007.

[Tur12] Volker Turau. Efficient transformation of distance-2 self-stabilizing algorithms. *Journal of Parallel and Distributed Computing*, 72(4):603–612, 2012.

[TW09] Volker Turau and Christoph Weyer. Fault tolerance in wireless sensor networks through self-stabilization. *International Journal of Communication Networks and Distributed Systems*, 2(1):78–98, 2009.

[Var93] George Varghese. *Self-stabilization by local checking and correction*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1993.

[Yam09] Yukiko Yamauchi. *A Study on Hierarchical Design of Fault-containing Self-stabilizing Protocols*. PhD thesis, Osaka University, 2009.

[YMB10] Yukiko Yamauchi, Toshimitsu Masuzawa, and Doina Bein. Adaptive containment of time-bounded byzantine faults. In *Stabilization, Safety, and Security of Distributed Systems*, volume 6366 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2010.

[YT10] Yukiko Yamauchi and Sébastien Tixeuil. Monotonic stabilization. In *Principles of Distributed Systems*, volume 6490 of *Lecture Notes in Computer Science*, pages 475–490. Springer, 2010.

List of Symbols

$\mathbb{N}$	set of positive integer numbers ($\mathbb{N}_0$ includes zero)
$\mathbb{Z}$	set of integer numbers
$\mathbb{Q}$	set of rational numbers
V	set of nodes (or processes)
E	set of edges (or communication links) $\subseteq V \times V$
n	number of nodes ($n = \|V\|$)
m	number of edges ($m = \|E\|$)
D	diameter of the topology $G = (V, E)$
Δ	maximum degree of the topology $G = (V, E)$
$\mathcal{A}$	a distributed algorithm
$\mathcal{I}_{\mathcal{A}}$	set of instances of $\mathcal{A}$, i.e., $\{(v, p) \mid v \in V \wedge p \in \mathcal{A}(v)\}$
$\sigma_{\mathcal{A}}$	set of all possible local states of $\mathcal{A}$
$\Sigma_{\mathcal{A}}$	set of all possible configurations of $\mathcal{A}$
$\Sigma_{\mathcal{A}}^{ext}$	set of all possible extended configurations of $\mathcal{A}$
$\mathcal{L}_{\mathcal{A}}$	legitimacy of configuration of $\mathcal{A}$ (Boolean predicate)
$\mathcal{L}_{\mathcal{A}}^{pri}$	legitimacy of primary configuration of $\mathcal{A}$ (predicate)
$\mathcal{S}_{\mathcal{A}}$	safety of configuration of $\mathcal{A}$ (Boolean predicate)
$c\|_m$	the local state of instance m at configuration c
$c \vdash e$	value of expression e at configuration c
$(c : m)$	result of a step under the central scheduler
$(c : S)$	result of a step under the distributed scheduler
$\exists!\, x \in X : p(x)$	there exists exactly one $x \in X$ satisfying $p(x)$
$\#\, x \in X : p(x)$	the number of elements $x \in X$ satisfying $p(x)$

List of Figures

List of Tables