
# Local Graph Transformation Primitives for Some Basic Problems in Overlay Networks

## Dissertation

In partial fulfillment of the requirements for the academic degree

**Doctor rerum naturalium (Dr. rer. nat.)**

at the Faculty of Computer Science,
Electrical Engineering and Mathematics
at Paderborn University

submitted by

**Alexander Setzer**

Paderborn, June 2020

**Advisor:**

Prof. Dr. Christian Scheideler

**Reviewers:**

Prof. Dr. Christian Scheideler
Prof. Dr. Friedhelm Meyer auf der Heide

**Members of the Doctoral Panel:**

| | |
|---|---|
| Prof. Dr. Christian Scheideler (Chairperson) | Paderborn University |
| Prof. Dr. Friedhelm Meyer auf der Heide | Paderborn University |
| Prof. Dr. Eckhard Steffen | Paderborn University |
| Dr. Matthias Fischer | Paderborn University |
| Dr. Saqib A. Kakvi | University of Wuppertal |

**Contact:**

Alexander Setzer (asetzer@mail.uni-paderborn.de)

# Abstract

This thesis considers some basic problems in overlay networks, focusing on the particular role that primitives for the transformation of the network graph play in their solution.

In the first part, we investigate the complexity of computing a minimum sequence of applications of graph transformation primitives to transform one graph into another. For this we consider two sets of primitives (one for undirected and one for directed graphs) that have proved to be useful in the context of self-stabilizing overlay networks. We find that the problem is NP-hard but are able to develop constant-factor approximation algorithms for both primitive sets.

In the second part, we study the problem of monotonic searchability in self-stabilizing overlay networks. We develop a general approach to guarantee monotonic searchability when the target topology is a supergraph of the line. Moreover, we present a protocol that solves monotonic searchability for the line topology and works correctly when nodes are allowed to leave the system as well. For our solutions, local graph transformation primitives turn out to serve as an important ingredient.

In the third part, we develop the relay model, a new model for interconnecting nodes. Whereas in the standard model nodes were not able to leave the system faithfully (i.e., without harming connectivity) unless they had access to an oracle, in the relay model these oracles are not required. We show how to implement the relay model in a self-stabilizing fashion and provide a generic approach to handle node departures in this model. Moreover, we prove the relay model to be universal. This means that arbitrary graph transformations are possible in this model. Even more, they are possible via (weak-)connectivity-preserving graph transformation primitives, which are "almost local" as they require the cooperation of up to three nodes (which is due to the fact that connections cannot be forwarded without consent in the relay model). Besides the aforementioned aspects, the relay model has several additional advantages. For example, protocols for the standard model can be transformed into the relay model. Furthermore, the model allows nodes to grant and revoke access rights, making it useful beyond the context of studying node departures.

# Zusammenfassung

Diese Dissertation betrachtet einige grundlegende Overlay-Netz-Probleme. Sie konzentriert sich dabei auf die besondere Rolle, welche Primitive zur Transformation des Netzwerkgraphen in ihrer Lösung spielen.

Im ersten Teil untersuchen wir die Komplexität der Berechnung einer kürzesten Abfolge von Anwendungen von Graphtransformationsprimitiven, um einen Graphen in einen anderen umzuformen. Hierfür betrachten wir zwei Mengen von Primitiven (eine für ungerichtete und eine für gerichtete Graphen), die sich im Kontext von Overlay-Netzen als nützlich erwiesen haben. Wir zeigen auf, dass dieses Problem zwar NP-schwer ist, können aber Approximationsalgorithmen mit konstantem Approximationsfaktor für beide Mengen von Primitiven angeben.

Im zweiten Teil behandeln wir das Problem der monotonen Suchbarkeit in selbststabilisierenden Overlay-Netzen. Wir entwickeln einen generellen Ansatz, um montone Suchbarkeit zu garantieren, wenn die Zieltopologie ein Obergraph der Linientopologie ist. Außerdem stellen wir ein Protokoll vor, welches das Problem der monotonen Suchbarkeit für die Linientopologie löst und selbst dann korrekt arbeitet, wenn Knoten das System verlassen dürfen. Lokale Graphtransformationsprimitive erweisen sich hierbei als wichtiger Bestandteil unserer Lösungen.

Im dritten Teil entwickeln wir das Relay-Modell, ein neues Modell zur Vernetzung von Knoten. Während es im Standard-Modell nicht möglich ist, dass Knoten das System gewissenhaft (das heißt, ohne den Zusammenhang zu gefährden) verlassen, sofern sie nicht Zugang zu einem Orakel haben, werden diese Orakel im Relay-Modell nicht benötigt. Wir zeigen, wie das Relay-Modell selbststabilisierend implementiert werden kann und stellen außerdem einen generellen Ansatz vor, um das Verlassen von Knoten in diesem Modell zu behandeln. Ferner zeigen wir, dass das Relay-Modell universell ist. Das bedeutet, dass beliebige Graphtransformationen in diesem Modell möglich sind. Genauer gesagt sind sie mithilfe von Graphtransformationsprimitiven möglich, die den (schwachen) Zusammenhang bewahren und "nahezu lokal" sind, da sie die Zusammenarbeit von bis zu drei Knoten erfordern (was damit zusammenhängt, dass Verbindungen im Relay-Modell nicht einfach weitergeleitet werden dürfen). Zusätzlich zu den bereits genannten Aspekten hat das Relay-Modell weitere Vorteile. So können beispielsweise Protokolle für das Standard-Modell in das Relay-Modell überführt werden. Außerdem ermöglicht letzteres, Zugriffsrechte zu gewähren und zu entziehen, wodurch es auch jenseits des Kontexts von Knotenabgängen von Nutzen ist.

# Contents

CHAPTER 1

# Introduction

One of the most basic tasks to be conducted in overlay networks is the transformation of the network graph. Even the joining or the departure of a single member of the distributed system may entail a large number of changes to the network. Apart from this, an overlay network may undergo continuous transformations for the purpose of optimizing certain metrics (such as dilation or congestion). In many cases, reaching a certain target topology is not the only goal, though. Often it is desirable to maintain some properties throughout the transformation process. For example, many applications require the network graph to be connected in every state. Other applications may also require certain monotonicity properties of the transformation, for instance to enable reliable search.

A major challenge for distributed systems is introduced by the fact that their members may display erroneous behavior. Unfortunately, for realistic, large systems, faults are not the exception but the norm. To approach this problem, two principles have turned out to be very helpful, which we will also follow in this thesis: The first is to perform the graph transformations by only *local* operations. This means that the changes induced by a single participant can affect only his neighborhood in the network graph. The second is to make (almost) no assumptions on the initial correctness of variables and messages in the design of all protocols: i.e., to make the system *self-stabilizing*.

An extremely useful tool in the context of local graph transformations in the self-stabilizing setting was introduced by Koutsopoulos, Scheideler, and Strothmann [KSS17]. The authors describe a set of simple *graph transformation primitives* that exhibit the following three desirable properties: First, they are local in the sense that the application of one such primitive by a node affects only the neighborhood of that node. Second, they preserve weak connectivity: i.e., none of their applications can disconnect a weakly connected component. Third, they are universal, which means that they can be used to transform a given weakly connected graph into any weakly connected graph consisting of the same nodes.

This thesis further investigates the graph transformation primitives due to Koutsopoulos, Scheideler and Strothmann as well as their usefulness for basic problems in the context of overlay networks. As an introduction, we study the complexity of computing a minimum sequence of graph transformation primitive applications to obtain a desired target graph. As it turns out, the problem is NP-hard but constant-approximable. In the further parts of the thesis we consider several problems in the self-stabilization setting.

The first problem in the self-stabilization setting we consider is the problem of searching reliably during the stabilization process: Simply put, if a message

from a node $u$ to a node $v$ succeeds at some point in time, then every future such message must do as well. Therefore, this problem is also called the problem of *monotonic searchability*. It turns out that an adaptation of the aforementioned graph transformation rules serves as an important ingredient to the solution of this problem.

The second problem in the self-stabilization setting we consider is that of safe departures. In general, nodes that want to leave the system cannot simply do so immediately: There could be a cut of the network graph consisting entirely of nodes that want to leave and their immediate departure would disconnect the network. For this reason, before the nodes wanting to leave have left the system, the network graph needs to be transformed such that all nodes can leave without harming connectivity. This problem was originally introduced by Foreback, Koutsopoulos, Nesterenko, Scheideler and Strothmann [For+14] and is known as the *finite departure problem*. The same authors also proved that the problem is not solvable without oracles in the standard communication model. In this thesis, we present an approach that solves monotonic searchability as well as the finite departure problem at the same time and assumes access to a reasonable oracle.

In the third part of this thesis, we design a novel interconnection model for overlay networks that permits a solution to the finite departure problem without oracles. To justify this model, we present a self-stabilizing implementation that only requires a reliable link-layer as a base. Our model possesses a number of interesting characteristics that make it useful for applications beyond the finite departure problem, e.g., in the access-control domain.

## 1.1. Four Basic Graph Transformation Primitives

We now describe the primitives for the manipulation of graphs, first introduced by Koutsopoulos, Scheideler, and Strothmann [KSS17] in the context of overlay networks.

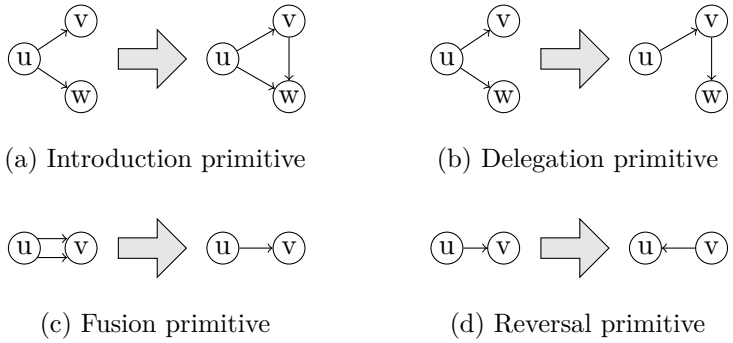
For directed graphs, consider the set $\mathcal{IDFR}$ consisting of the following four graph transformation primitives:

**Introduction** If a node $u$ has a reference of two nodes $v$ and $w$ such that $v \neq w$, $u$ *introduces* $w$ to $v$ if $u$ sends a message to $v$ containing a reference of $w$ while keeping the reference.

**Delegation** If a node $u$ has a reference of two nodes $v$ and $w$ such that $u, v, w$ are all different, then $u$ *delegates* $w$'s reference to $v$ if $u$ sends a message to $v$ containing a reference of $w$ and deletes the reference of $w$.

**Fusion** If a node $u$ has two references $v$ and $w$ such that $v = w$, then $u$ *fuses* the two references if it only keeps one of these references.

**Reversal** If a node $u$ has a reference of some other node $v$, then $u$ *reverses* the connection if it sends a reference of itself to $v$ and deletes its reference of $v$.




(a) Introduction primitive (b) Delegation primitive

(c) Fusion primitive (d) Reversal primitive

Figure 1.1.: The four primitives in $\mathcal{IDFR}$ in pictures.

The four primitives are visualized in Figure 1.1. Note that for the introduction primitive, it is possible that $w = u$: i.e., $u$ introduces itself to $v$. To simplify the description, we sometimes say that a node $u$ introduces or delegates the *edge* $(u, v)$ if $u$ introduces $v$ to some other node or delegates $v$'s reference to some other node, respectively.

The primitives in $\mathcal{IDFR}$ are known to be universal (c.f. [KSS17]): i.e., it is possible to transform any weakly connected graph into any other weakly connected graph by using only the primitives in $\mathcal{IDFR}$. Note that for every edge $(u, v)$ used in any of the primitives, either $(u, v)$ still exists after the corresponding primitive is applied, or there is still an (undirected) path from $u$ to $v$ in the resulting graph. This directly implies that no application of the primitives can disconnect the graph.

For undirected graphs, consider the set $\mathcal{IDF}$ containing only the primitives introduction, delegation and fusion (defined correspondingly). These three primitives, accordingly, are universal on undirected graphs: i.e., any connected undirected graph can be transformed into any other connected undirected graph by applying the primitives in $\mathcal{IDF}$ (c.f. [KSS17]).

## 1.2. Motivation

Despite being quite simple and fully local, the graph transformation primitives described in Section 1.1 become very powerful because they preserve (weak) connectivity and are universal. Therefore, it is reasonable to investigate these primitives further. This thesis does so by addressing the following three central questions: First, what is the complexity of transforming graphs with a minimum number of primitive applications? Second, in which way can the primitives (and suitable adaptations of them) be used to develop self-stabilizing protocols for network topologies that, in addition to the pure stabilization, solve the problem of monotonic searchability? Third, how is it possible to overcome the issue inherent to the primitives that they might violate user rights (caused by the fact that references of nodes can be forwarded without permission)?

## 1.3. Related Work

This section gives an overview of the relevant existing research in the subject areas of this thesis. Since every part of this thesis is in some way concerned with primitives for transforming graphs, we begin in Section 1.3.1 with giving an overview of the existing literature regarding graph transformations in distributed systems. The majority of this thesis deals with self-stabilizing systems, more precisely self-stabilizing overlay networks. In Section 1.3.2 we thus give an overview of the related work in this area. This includes related work on the specific problems (such as monotonic searchability and the finite departure problem) concerned in this thesis. The third part of this thesis deals with a new model for the interconnection of nodes in distributed systems based on so-called relays. As a matter of fact, there are many models or concepts that aim at overcoming some of the issues our relay model solves. Therefore, in Section 1.3.3 we give an overview of similar or related concepts. Last, in Section 1.3.4 we list additional literature that is related to this thesis in some way.

### 1.3.1. Graph Transformations in Distributed Systems

Graph transformations have been studied in many different contexts and applications, including but not limited to pattern recognition, compiler construction, computer-aided software engineering, the description of biological developments in organisms and the implementation of functional programming languages. Detailed introductions and an extensive overview of the existing literature in this area even beyond distributed systems can be found in [And+99], [Hec06], or [Roz97; Ehr+99], for example. Simply put, a graph transformation (or graph-rewriting) system consists of a set of rules $L \to R$ that may be applied to subgraphs isomorphic to $L$ of a given graph $G$, thus replacing $L$ with $R$ in $G$. One can generate a graph grammar by enumerating all graphs reachable from some starting graph by the (repeated) application of these rules. These graph grammars can also be viewed as generalizations of Chomsky grammars (see [ASK18]). Since changing the labels assigned to a graph (graph relabelling) is also a kind of graph transformation, basically every distributed algorithm can be understood as a graph transformation system (c.f. [Ehr+99]).

Transforming the network graph is one of the most basic tasks to be carried out in overlay networks. The purpose of network graph transformations is not only to integrate new nodes into the network or to remove nodes from them. In fact, topologies are often transformed continuously to optimize certain metrics, for example to ensure connectivity properties of the graph [Liu+06] or to perform load-balancing [Kru+10]. The idea of transforming the network graph according to a set of rules, as employed in Chapter 3, is not entirely new. In fact, Stein et al. [Ste+16] proposed TARL, a language for expressing topology adaptation rules. These rules can be far more complex than the rules considered in this thesis. In contrast to our work, however, that work focuses on empirical evaluations and does

not consider the complexity of transforming a graph with a minimum number of rule applications.

Another interesting problem related to our work in Chapter 3 is the so-called *graph transformation problem* [Lin94]. In this problem the goal is to find the minimum integer $k$ such that an initial graph $G_s$ can be transformed into a final graph $G_t$ by relocating at most $k$ edges in $G_s$. Lin proved that the decision version of the graph transformation problem for general $k$ is NP-complete [Lin94]. In contrast to our work, the nodes in this setting are not uniquely identifiable, causing, for example, the decision version of this problem for $k = 0$ to be the graph isomorphism problem. Another difference is that in this thesis we do not allow arbitrary edge relocations but restrict them to a set of rules that can be applied locally.

### 1.3.2. Self-Stabilization and Self-Stabilizing Overlay Networks

Self-stabilization in distributed computing was originally introduced by Dijkstra in 1974 [Dij74]. In this seminal work, he considered the synchronization of machines arranged in a ring topology. Since then, self-stabilization has been considered in many applications, such as clock synchronization, model conversion and monitoring (see [Dol00] for an overview). This thesis is specifically concerned with self-stabilizing overlay networks. The problem of stabilizing network topologies from arbitrary weakly connected states has first been considered for simple line and ring networks (e.g., [SR05; ORS07; Gal+14]). Over the years increasingly more network topologies were considered, ranging from skip lists and skip graphs [NNS13; Jac+14] to expanders [DT13], Delaunay graphs [Jac+12], double-headed radix trees [AW07] hypertrees [DK08], De Bruijn networks [RSS11], small-world graphs [KKS12], and variants of Chord [KKS14]. Furthermore, a universal algorithm for topological self-stabilization has been developed, known as the Transitive Closure Framework [BGP13]. Only recently, Feldmann, Schmid, and Scheideler published a thorough survey on self-stabilizing overlay networks [FSS20].

The four graph transformation primitives introduced by Koutsopoulos, Scheideler, and Strothmann, which in some way play a role throughout this thesis, have originally been introduced for the topic of self-stabilizing overlay networks as well [KSS17]. The authors found out that three of these primitives (introduction, delegation and fusion) are sufficient to turn any weakly connected graph $G$ into any strongly connected graph $G'$ and that all four primitives together are sufficient to achieve the same for weakly connected graphs $G'$. In addition, they proved that the four primitives are also necessary to accomplish these graph transformations in general.

The notion of monotonic searchability, which Part II considers, was introduced by Scheideler, Setzer and Strothmann [SSS15]. Apart from defining monotonic searchability, the authors proved in that paper that it is impossible to satisfy monotonic searchability if arbitrary corrupted messages are present, and they showed how to achieve monotonic searchability in the line topology. The paper also

has a second part, in which monotonic searchability is solved in the setting where nodes may leave the system, and this part serves as the basis for Chapter 5 of this thesis. After that, Scheideler, Setzer, and Strothmann considered a universal approach for monotonic searchability [SSS16]. Since this approach can entail a high message cost, researchers still considered monotonic searchability for particular topologies afterwards. For example, Feldmann, Kolb, and Scheideler developed a self-stabilizing quad-tree protocol satisfying monotonic searchability [FKS18]. Moreover, Luo, Scheideler, and Strothmann considered monotonic searchability for the perfect skip graph [LSS19]. Closely related to monotonic searchability is the notion of *monotonic convergence* by Yamauchi and Tixeuil [YT10]. A self-stabilizing protocol is monotonically converging if every change done by a node $p$ makes the system approach a legal state and if every node changes its output only once. The authors investigate monotonically converging protocols for different classic distributed problems (e.g., leader election and vertex coloring) and focus on the amount of non-local information that is needed for them. Even apart from that, maintaining safety properties during the convergence phrase is not entirely new. In fact, several such approaches have been investigated in self-stabilization. One example of this is snap-stabilization [Bui+07; Del+10]: A protocol is *snap-stabilizing* if it always behaves according to its specification independent of its initial configuration. Another example is super-stabilization [DH97]: A *super-stabilizing* protocol guarantees that starting from a legal configuration, a safety property is preserved after only one specific topology change and that the safety property is maintained during the convergence to a legal configuration, assuming that no more topology changes occur during the stabilization phase. In contrast to super-stabilization, *self-stabilization with service guarantee* [JM10] provides and maintains the safety property even before the stabilization. As a generalization of super-stabilization, safe convergence [KM06] was formulated. When the *safe convergence* property is fulfilled, the system quickly converges to a configuration fulfilling a safety property and continues to fulfill that safety property while converging to a legal configuration.

The *finite departure problem* ($\mathcal{FDP}$) studied in two main chapters of this thesis was originally introduced by Foreback et al. [For+14]. The aim was to investigate graceful departures of nodes in a self-stabilizing setting. Simply put, in the $\mathcal{FDP}$, nodes that want to leave a distributed system should decide when they can leave without affecting the weak connectivity of the topology. The authors of [For+14] conclude that it is not possible to solve the $\mathcal{FDP}$ in general. However, with the use of distributed oracles (which are specialized failure detectors [CT96]) the authors propose a protocol that solves the problem and arranges the nodes in a line. Additionally, they show that oracles are not needed if the problem is transformed into a non-decision variant (called the *finite sleep problem*). In this problem, nodes just fall asleep whenever they think it is safe to do so, but they will be woken up again whenever a message is delivered to them. Therefore, the goal of the finite sleep problem is just to ensure that eventually a state is reached at which all leaving nodes are permanently asleep. In the aftermath, Koutsopoulos, Scheideler, and

Strothmann generalized the idea of the $\mathcal{FDP}$ to a protocol framework that solves the $\mathcal{FDP}$ without being reliant on a certain topology [KSS17]. This protocol is thus combinable with most existing overlay protocols. The idea of this framework also influenced the results of Chapter 6 where we present a framework to solve the $\mathcal{FDP}$ in the relay model. A significant difference, though, is caused by the fact that the oracle used in [KSS17] is not implemented by the relay layer, which is why it was not possible to simply transfer that result for this thesis.

### 1.3.3. Relays

The idea of using relays in communication networks, as we do in Chapter 6 of this thesis, is not entirely new, but has a long history already. Relays are commonly used when two devices are too far away from each other to exchange information directly, like in wireless networks, or if two devices cannot interact directly because of firewalls. Relay networks have also been used to improve availability (prominent examples are Resilient Overlay Networks [And+01]), to provide anonymity (a prominent example is the TOR network [DMS04]) or to improve performance (a prominent example is AKAMAIs IPA Relay service). In general, most of the peer-to-peer systems and overlay networks proposed so far are using their members as relays for the exchange of requests or information between its members.

The relay concept we propose in Chapter 6 has some interesting connections to the access control domain. As we will see, our concept is equipped with commands to close a relay or to shut down a process (called delete and stop, respectively). This makes it possible to grant and revoke access rights with relays. There is a substantial amount of literature on access control in distributed systems. For surveys on access control approaches in various contexts such as operating systems, file systems, distributed systems, and web-based systems, we refer to [Del+07; KMD17; Mil+08; HFK06; Kos09; LMM10]. One can summarize that there are three important requirements for access control schemes: integrity, propagation, and revocation. *Integrity* means that it should not be possible to construct, tamper with or steal an access right. *Propagation* means that there should be mechanisms in place controlling the transfer of access rights. *Revocation* means that it should be possible to revoke an access right. Interestingly, the relay approach presented in Chapter 6 can satisfy these requirements if the processes cannot tamper with the relay layer introduced there.

The simplest ways of controlling access rights are to use passwords or cryptographic keys, but these can easily be delegated from one process to another. Another simple method is to use access control lists (ACLs), which gained prominence in the 1970s with the advent of multiuser systems such as UNIX. In distributed systems, the ACL approach usually requires a trusted third party, in order to prevent tampering with the ACLs. Other popular access control models are Role-Based Access Control (RBAC), Attribute-Based Access Control (ABAC), Policy-Based Access Control (PBAC), and Risk-Adaptive Access Control (RAdAC). For all of these models, various variants have been proposed depending on the context in

which they are used and the focus on particular properties. In most of the implementations of these models, trusted third parties are used as well since otherwise it is hard to guarantee integrity and prevent uncontrolled propagation of access rights. More decentralized approaches keep track of delegation chains, which is somewhat similar to our relay approach: If one of the delegations is revoked, all delegations beyond it will not be accepted any more so the corresponding processes will lose their access rights to certain objects. However, to the best of our knowledge, this chaining approach has not been used in order to control the interconnection of processes. An example where access rights are provided via explicit communication channels is the Singularity operating system [Hun+05; Wob+07]. A key aspect of Singularity are Software-Isolated Processes (SIPs), which encapsulate pieces of an application or a system and provide information hiding, failure isolation, and strong interfaces. Communications between SIPs is through bidirectional, strongly typed, higher-order channels. When a channel is created, both of its endpoints are returned to the SIP that created it. These endpoints can be freely delegated along existing channels but not replicated, which provides a more flexible form of access control than our relay approach, but still opens up the possibility of stealing access rights or delegating them by mistake.

For the relay model proposed in this thesis, we assume a reliable link layer. Therefore, the research by Dolev et al. on self-stabilizing link layers [Dol+11; Dol+12] is noteworthy. They provide self-stabilizing algorithms to deliver (higher level) messages in FIFO order without duplicates or omissions even if the network may duplicate, omit or reorder packets. The assumption required for this is that the total number of packets in the system at every point in time is bounded. Their protocol may be applied below the relay layer we propose, to obtain an eventually-reliable link layer. We highlight, however, that the FIFO assumption is not required for the relay model.

### 1.3.4. Additional Literature

In thesis (more specifically, in Chapter 3), we prove the NP-hardness of two problems and provide approximation algorithms for them. For the NP-hardness proofs, we conduct a reduction from the well-known satisfiability problem (SAT), whose NP-hardness was proven independently by Cook [Coo71] and Levin [Lev73] and represents one of the most fundamental results in theoretical computer science. For the approximation algorithms, we use an approximation algorithm for the undirected Steiner forest problem as a black-box. In this problem, a graph $G$ and a set $S$ of pairs of nodes from $G$ are given. The task is to find a forest $F$ in $G$ of minimum cost such that in $F$ the two nodes of each pair in $S$ are connected by a path. The cost of the forest is the sum of the edge costs, which are defined according to a metric. This problem is also known as the Steiner subgraph problem with edge sharing. Since the Steiner forest problem is a generalization of the Steiner tree problem, in which the goal is to find a tree connecting a set of given terminals, its NP-hardness is implied by the NP-hardness of the Steiner tree. The

latter was proven by Karp [Kar72]. In a generalization of the Steiner forest problem called the survivable network design problem or the generalized Steiner problem, the input may additionally consist of a connectivity requirement for each pair of nodes, i.e., a number of distinct paths connecting the pair of nodes. According to a survey by Kerivin and Mahjoub [KM05], this generalized problem was introduced by Steiglitz, Weiner, and Kleitman [SWK69]. The first 2-approximations of this problem were primal-dual algorithms given by Agrawal, Klein, and Ravi [AKR95] and by Goemans and Williamson [GW95] who generalized the former results. Later, Jain [Jai01] also presented a 2-approximation using an iterative rounding technique. Gupta and Kumar [GK15] showed a simple greedy algorithm to have a constant approximation ratio. Recently, Groß et al. [Gro+18] presented a local-search constant approximation for the Steiner forest. On a side note, the Steiner forest problem was also considered in the online setting: Awerbuch, Azar, and Bartal showed a greedy algorithm to be $O(\log^2 |S|)$-competitive [AAB04], and Berman and Coulston gave an $O(\log |S|)$-competitive algorithm [BC97].

## 1.4. List of Own Publications

In addition to the publications that directly serve as a basis for the main chapters of this dissertation, I co-authored a number of other publications as well during my research time at Paderborn University. In one way or another, my work on these topics also influenced this dissertation, which is why I will give a brief overview of the published works I participated in so far.

Aside from the topics of this thesis, I additionally worked on game theory (see [Cor+12]), online algorithms (in particular, the minimum linear arrangement problem, see [ESS14a]), and robust distributed storage systems (see [ESS14b]). Furthermore, I worked on a distributed queue (see [FSS18]) and the consensus problem (see [RSS18]). Additional research involved multi-dimensional range queries in peer-to-peer based storage (see [Ben+18]) and a minimum spanning tree construction on tree-underlays (see [GSS18]).

The complete list of papers co-authored by me and published so far is the following:

[Cor+12] Andreas Cord-Landwehr, Martina Hüllmann, Peter Kling, and Alexander Setzer. **Basic Network Creation Games with Communication Interests**. In: *Proceedings of the 5th International Symposium on Algorithmic Game Theory (SAGT)*. Barcelona, Spain, 2012.

[ESS14a] Martina Eikel, Christian Scheideler, and Alexander Setzer. **Minimum Linear Arrangement of Series-Parallel Graphs**. In: *Revised Selected Papers of the 12th International Workshop on Approximation and Online Algorithms (WAOA)*. Wrocław, Poland, 2014.

[ESS14b] Martina Eikel, Christian Scheideler, and Alexander Setzer. **RoBuSt: A Crash-Failure-Resistant Distributed Storage System**. In: *Proceed-*

*ings of the 18th International Conference on Principles of Distributed Systems (OPODIS)*. Cortina d'Ampezzo, Italy, 2014.

[SSS15] Christian Scheideler, Alexander Setzer, and Thim Strothmann. **Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures**. In: *Proceedings of the 19th International Conference on Principles of Distributed Systems (OPODIS)*. Rennes, France, 2015.

[SSS16] Christian Scheideler, Alexander Setzer, and Thim Strothmann. **Towards a Universal Approach for Monotonic Searchability in Self-stabilizing Overlay Networks**. In: *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*. Paris, France, 2016.

[FSS18] Michael Feldmann, Christian Scheideler, and Alexander Setzer. **Skueue: A Scalable and Sequentially Consistent Distributed Queue**. In: *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Vancouver, British Columbia, Canada, 2018.

[RSS18] Peter Robinson, Christian Scheideler, and Alexander Setzer. **Breaking the $\tilde{\Omega}(\sqrt{n})$ Barrier: Fast Consensus under a Late Adversary**. In: *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Vienna, Austria, 2018.

[Ben+18] Markus Benter, Till Knollmann, Friedhelm Meyer auf der Heide, Alexander Setzer, and Jannik Sundermeier. **A Peer-to-Peer Based Cloud Storage Supporting Orthogonal Range Queries of Arbitrary Dimension**. In: *Revised Selected Papers of the 4th International Symposium on Algorithmic Aspects of Cloud Computing (ALGOCLOUD)*. Helsinki, Finland, 2018.

[GSS18] Thorsten Götte, Christian Scheideler, and Alexander Setzer. **On Underlay-Aware Self-Stabilizing Overlay Networks**. In: *Proceedings of the 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Tokyo, Japan, 2018.

[SS18] Christian Scheideler and Alexander Setzer. **Relays: A New Approach for the Finite Departure Problem in Overlay Networks**. In: *Proceedings of the 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Tokyo, Japan, 2018

[SS19] Christian Scheideler and Alexander Setzer. **On the Complexity of Local Graph Transformations**. In: *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP)*. Patras, Greece, 2019.

## 1.5. Contribution and Outline of the Thesis

Chapter 2 contains preliminaries required for the four main chapters of this thesis. It gives an overview of the models used in this thesis and some notions respective to them, formally defines all problems that play a role in more than one chapter, and explains the pseudocode notation.

The thesis is divided into three parts: The first deals with the complexity of the aforementioned local graph transformations, the second with monotonic searchability and the third with the novel interconnection model for the finite departure problem.

Part I consists of Chapter 3, which deals with the complexity of computing a minimum sequence of applications of the graph transformations due to Koutsopoulos, Scheideler and Strothmann. We show that this problem is NP-hard in both the undirected and the directed case. For both cases, we also present approximation algorithms that have a constant approximation ratio.

Part II consists of two chapters that both deal with monotonic searchability. In Chapter 4, we provide a general framework to enable monotonic searchability in a wide range of self-stabilizing topologies thereby generalizing a previous result by Scheideler, Setzer and Strothmann (first part of [SSS15]). For this approach, we use adaptations of the graph transformation primitives mentioned before. Our framework not only generally enables monotonic searchability but also allows successful search requests to reach their target fast (where the exact definition of "fast" depends on the topology). As a small digression, for a special class of graphs, called the $SKIP^+$ graphs, we show that monotonic searchability can alternatively be obtained by lifting the requirements for the topology that should be formed: If one allows the final topology to be a superset of the actual topology, monotonic searchability can be enabled by only a slight adaptation of an existing protocol for the self-stabilization of the $SKIP^+$ graph. In Chapter 5 we then present a protocol that enables monotonic searchability and solves the finite departure problem at the same time. Because of the description complexity involved with this, this protocol is not a general framework for a set of topologies, but a specific solution for a simple class of graphs, namely the line graph.

Part III consists of Chapter 6, in which we introduce a new interconnection model called the *relay model*. As we show in this chapter, the relay model has a number of advantageous properties. First of all, it allows transforming arbitrary graphs into arbitrary other graphs. Furthermore, existing protocols for the classical model can easily be adapted for the relay model. Moreover, only assuming a reliable or self-stabilizing link layer, the relay model can be implemented in a self-stabilizing fashion: i.e., even if the variables and messages for the internal operation of the relay layer are initially corrupted, the relay layer will eventually conform to its specification.[1] Last, but perhaps most importantly since this originally motivated

[1] This is a simplified view, for the ease of description. In fact, some additional assumptions need to be made, but this will be clarified in the corresponding chapter.

the invention of the relay model, it is possible to solve the finite departure problem without an oracle in this model. We prove this by developing and analyzing a general protocol that can be used to transform existing protocols in the relay model such that nodes can leave the system without harming connectivity, i.e., such that, in addition to their original behavior, they solve the finite departure problem. It turns out that the relay model has the potential to be useful in other scenarios as well, such as in the access control domain, which we will discuss in the conclusion.

The last part of this thesis is Part IV. It consists of Chapter 7 where we review the results of this thesis and give an outlook on open research questions raised by these results.

CHAPTER 2

# Preliminaries

This chapter lays the technical foundations for this thesis. Here we explain the graph transformation primitives that this thesis builds on, introduce the main models considered in this thesis and define some terms used throughout the thesis. We also formally define various problems that are used in several chapters of the thesis. Specific problems that are relevant to single chapters only are defined in the respective chapters. Moreover, at the end of this chapter, we explain the pseudocode notation used in this thesis.

**Outline of This Chapter** In Section 2.1, we give an overview of the models used in this thesis. After that, Section 2.2 introduces some additional terminology relevant for the whole thesis and restates some existing results on the graph transformation primitives in $\mathcal{IDF}$. The formal definition of the problems referred to extensively in this thesis is then given in Section 2.3. Last, in Section 2.4 we explain how to read the pseudocode used in most chapters of this thesis.

## 2.1. Model Overview

In this section, we describe the models used in this thesis. For the complexity analysis in Part I, it is sufficient to consider a very simple system model, which we describe in Section 2.1.1. For the problems in the self-stabilization setting (Part II and Part III), we consider a more complex system model, which we describe in Section 2.1.2.

### 2.1.1. System Model for the Complexity Analysis

For Part I of this thesis, we consider a simple model. We model the overlay network as a graph: i.e., nodes represent participants of the network and if there is a directed edge $(u, v)$ in the graph, this means that there is a connection from $u$ to $v$. Undirected edges $\{u, v\}$ model the two connections from $u$ to $v$ and from $v$ to $u$. Since there may be multiple connections between the same pair of participants, the graphs we consider in this thesis are multigraphs: i.e., edges may appear several times in the multiset of edges. For convenience, throughout this thesis we will use the term "graph" instead of "multigraph" and refer to the "edge set" of a graph even though it is actually a multiset.

In this setting, a *computation* $C$ is a finite sequence $G_1 \to G_2 \to \ldots \to G_l$ of either directed or undirected graphs, in which each graph $G_{i+1}$ is obtained from $G_i$ by the application of a single primitive from $\mathcal{IDFR}$ or $\mathcal{IDF}$, respectively. The

graphs $G_1$ and $G_l$ are called the *initial* and the *final* graphs of $C$, respectively. The variable $l$ is called the *length of the computation*.

### 2.1.2. System Model for Monotonic Searchability and Relays

The system model used for Part II and Part III is as follows: We consider a distributed system consisting of a fixed set of nodes that are interconnected to each other and in which each node has a unique immutable numerical identifier. The nodes are controlled by a local-control protocol that specifies the variables and actions that are available at each node. We distinguish between protocol-based variables that are defined along with the protocol and system-based variables defined by the communication model.

We now describe the computation model used in this setting. Since we consider two different communication models for Part II and Part III, we specify them in the corresponding parts of this thesis.

#### Computation Model for Monotonic Searchability and Relays

There are two types of *actions* that a protocol can execute. The first type has the form of a standard procedure consisting of a header $\langle label \rangle(\langle parameters \rangle)$ and a body $\langle commands \rangle$, where *label* is the unique name of that action, *parameters* specifies the parameter list of the action, and *commands* specifies the commands to be executed when calling that action. Such actions can be called locally (which causes their immediate execution) or remotely. In fact, we assume that every message must have the form $\langle label \rangle(\langle parameters \rangle)$, where *label* specifies the action to be called in the receiving node and *parameters* contains the parameters to be passed to that action call. All other messages are ignored by the nodes. The second type is a dedicated action called the *timeout* action. Its header simply consists of the keyword TIMEOUT and its body is a sequence of commands, too.

The *system state* is an assignment of values to every variable of each node. An action in some node $v$ is *enabled* in some system state if and only if it is a timeout action or there is a message requesting to call it (with further details on this to be specified in the communication model).

In this setting, a *computation* is an infinite sequence of system states such that for each state $S_i$, the next state $S_{i+1}$ is obtained by executing an action that is enabled in $S_i$. This disallows the overlap of action executions: i.e., action executions are *atomic*. We assume *weakly fair action execution*, meaning that if an action is enabled in all but finitely many states of a computation, then this action is executed infinitely often. Note, in particular, that the timeout action of a node is executed infinitely often. Besides this, we place no bounds on node execution speeds and no restrictions on the execution order of enabled actions: i.e., we allow fully asynchronous computations and non-FIFO message delivery.

## 2.2. Further Known Results and Additional Terminology

Regarding the set $\mathcal{IDF}$ consisting of the three primitives introduction, delegation and fusion, we can make the following observation:

**Observation 2.1.** *The introduction primitive is the only primitive that can increase the number of edges in a graph. The fusion primitive is the only primitive that can decrease the number of edges in a graph. The delegation primitive is the only primitive that can remove the last edge between two nodes (i.e., an edge of multiplicity one).*

When nodes can perform only local computations, it seems intuitive that the network graph is transformed in a way similar to applying one of these graph transformation primitives. In fact, the following was proven in [SSS16]:

**Theorem 2.2.** *Any compare-store-send protocol that self-stabilizes to a static strongly-connected topology and preserves weak connectivity can be transformed such that the interactions between nodes can be decomposed into the primitives of $\mathcal{IDF}$.*

In Part II and Part III it will become useful to have the following three terms defined that deal with computations and states in the computation. As these terms are not used in Part I, we only define them with respect to the definition of a computation according to Section 2.1.2:

**Definition 2.3** (Reachable State)**.** *We say a state $S'$ is reachable from a state $S$ if and only if there is a sequence of possible action executions such that beginning in state $S$, the resulting state is $S'$.*

**Definition 2.4** ($S > S'$, $S \geq S$)**.** *When the computation we refer to is clear from the context, we use the notion $S' > S$ as a shorthand to indicate that the state $S$ happened chronologically before $S'$. Accordingly, $S' \geq S$ means that $S' > S$ or $S' = S$.*

**Definition 2.5** (Computation Suffix)**.** *For a computation $C$ and a state $S$ of $C$, the* computation suffix *$SUFFIX_C(S)$ is the sub-sequence of $C$ starting with $S$. Where it is clear from the context, we might omit the $C$ and just refer to the suffix by $SUFFIX(S)$.*

Notice that according to the above definition, a computation suffix is also a computation.

## 2.3. Formal Problem Definitions

We now define all problems that are relevant for more than one chapter. Every problem that is relevant for a single chapter only will be defined in the corresponding chapter.

### 2.3.1. Self-Stabilization

In Chapters 4 to 6, we consider so-called self-stabilizing protocols. A protocol is *self-stabilizing* if it satisfies the following two properties as long as no transient faults occur:

**Convergence** Starting from an arbitrary system state the protocol is guaranteed to arrive at a legal state.

**Closure** Starting from a legal state the protocol remains in legal states thereafter.

A self-stabilizing protocol is thus able to recover from transient faults regardless of their nature. Moreover, a self-stabilizing protocol does not have to be initialized as it eventually starts to behave correctly regardless of its initial state.

### 2.3.2. Monotonic Searchability

Part II deals with searching during the stabilization process in self-stabilizing topologies. We model searching for nodes in the network in the following way: Whenever a node $v$ wants to search for another node, it initiates a $\textsc{Search}(v, destID)$ message in which $destID$ is the identifier of the node searched for. Here, $destID$ does not need to be an identifier of an existing node since it is also possible to search for a node that is not in the system. Every such $\textsc{Search}(v, destID)$ message is then routed through the network according to some *search protocol* $S_P$. This search protocol is required to deliver the $\textsc{Search}(v, destID)$ message to the node $w$ with $id(w) = destID$ if it exists or to drop the message otherwise, both within a finite number of steps. In the former case, we say the search request *succeeds*. In the latter case, we say it *fails*.

Traditionally, search protocols for a given topology were only required to deliver the search messages reliably once a legal state has been reached. However, in self-stabilizing systems, it is never possible to determine whether a legal state has been reached. Furthermore, searching reliably during the stabilization phase is much more involved. To formalize this, we introduce the following notion of *monotonic searchability*:

**Definition 2.6** (Monotonic Searchability)**.** *A self-stabilizing protocol $P$ satisfies* monotonic searchability *according to some search protocol $P_S$ if search requests are routed according to $P_S$ and it fulfills the following two properties for every computation $C$ of $P$:*

***Monotonicity*** *For every pair of nodes $v, w$ and every successful $\textsc{Search}(v, id(w))$ request $r$ initiated in some state $S$ of $C$, every $\textsc{Search}(v, id(w))$ request $r'$ such that $r'$ is initiated in a state $S' \geq S$ also succeeds.*

***Non-Triviality*** *$C$ has a suffix such that in this suffix for every pair of nodes $v, w$, $\textsc{Search}(v, id(w))$ requests will succeed if there is a path from $v$ to $w$ in the target topology.*

We do not mention $P_S$ if it is clear from the context. Note that the *non-triviality* property is required because a search protocol that never delivers any message would be one according to which every protocol would trivially satisfy the monotonicity property, but such a protocol is not desired.

### 2.3.3. The Finite Departure Problem

In Chapter 5 and Chapter 6, we consider departures of nodes from the network. To model the ability of a node to leave the system, we let each node have a variable $mode \in \{\text{leaving}, \text{staying}\}$ that can only be changed from *staying* to *leaving* (and not vice-versa). If for a node $u$, $u.mode = leaving$, the node is *leaving*; otherwise, the node is *staying*. Note that staying nodes can dynamically decide at any arbitrary state if they want to leave the system by changing the value of their *mode* variable to *leave*. To actually depart from the system, a leaving node may execute a dedicated **exit** command. When a node executes **exit** all remaining edges to or from that node are deleted and the node is further referred to as *inactive*. For an inactive node all actions are disabled and, in particular, it will not execute the timeout action regularly. Any node that is not inactive (be it staying or leaving) is called *active*. This way of modelling node departures is common in the literature (see, e.g., [For+14; KSS17]). The problem gets challenging when we want leaving nodes to be excluded from the system in a faithful manner: i.e., their departure must not disconnect the network. To model this formally, Foreback, Koutsopoulos, Nesterenko, Scheideler and Strothmann introduced the *finite departure problem* [For+14]. We state this problem in a slightly adapted way as follows:

**Definition 2.7** (Finite Departure Problem ($\mathcal{FDP}$))**.** *In case the **exit** command is available, eventually reach a system state such that:*

1. *every staying node is active,*
2. *every leaving node is inactive, and*
3. *for each weakly connected component of the initial network graph, the staying nodes in that component still form a weakly connected component.*

## 2.4. Pseudocode Explanation

We now briefly explain the notation of the pseudocode provided in Part II and Part III. These parts use the computation model described in Section 2.1.2. For the explanation we will refer to the following short pseudocode:

```
1  INTRODUCE(u)
2    N := N ∪ {u}
3
4  TIMEOUT
5    for all u ∈ N do
6      send INTRODUCE(self) to u
```

Line 1 contains the header of an action named INTRODUCE()[1] that has one parameter named $u$. This means that the indented lines following this line contain the actions to be executed when this action is called, i.e., when the executing node receives a message of type INTRODUCE(). Whenever a node accesses its own variables, we simply use the name of that variable in the pseudocode. Therefore, Line 2 means that the value of parameter $u$ is added to the (set) variable $N$ owned by the executing node.

Line 4 is the header of the TIMEOUT action that is executed periodically. In Line 5 one can see an example of a typical pseudocode construct (a *for* loop, in this example). These constructs (such as conditional statements marked by *if* and *else* keywords) are very common in computer science and thus will not be explained in greater detail here. In Line 6, the executing node sends an INTRODUCE() message. The parameter of this message is a reference of the executing node itself, represented by the special keyword $self$. The receiver of the message is the node whose reference is stored in the variable $u$. Here it is implicitly assumed that all elements in $N$ are references of nodes. To simplify the description, we may also refer to the node whose reference is stored in the variable $u$ by $u$ where this does not cause any ambiguities.

[1] Throughout this thesis, we add parentheses to the name of every action but the TIMEOUT action to indicate that it is an action.

PART

# I

# The Complexity of Local Graph Transformations

The first main part of this thesis reconsiders the universal graph transformation primitives $\mathcal{IDF}$ and $\mathcal{IDFR}$ introduced by Koutsopoulos, Scheideler and Strothmann and answers the following question: *What is the complexity of transforming arbitrary initial graphs into arbitrary final graphs using $\mathcal{IDF}$ and $\mathcal{IDFR}$?*

We not only prove that computing minimum transformation sequences is NP-hard, but also provide approximation algorithms for this problem that have a constant approximation factor.

CHAPTER 3

# NP-Hardness and Approximability of Local Graph Transformations

In this chapter, we analyze the complexity of computing a minimum sequence of steps to transform arbitrary graphs into each other, when in each step, the application of a single primitive from $\mathcal{IDFR}$ (for directed graphs) or $\mathcal{IDF}$ (for undirected graphs) is allowed.

Via a reduction from the classical satisfiability problem we prove that these two problems are NP-hard. However, we additionally show that both of them can be constant-approximated.

The main results of this chapter have previously appeared in the following publication:

> Christian Scheideler and Alexander Setzer. **On the Complexity of Local Graph Transformations**. In: *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP)*. Patras, Greece, 2019. [SS19]

**Outline of This Chapter** We first formally state the problems considered in this chapter in Section 3.1. The rest of this chapter consists of two main parts: In Section 3.2, we prove the undirected and the directed problem variant to be NP-hard. In Section 3.3 we then present constant approximation algorithms for the two problems.

## 3.1. Problem Statement

We formally define the *Directed Local Graph Transformation Problem* (DLGT) and the *Undirected Local Graph Transformation Problem* (ULGT) as follows:

**Definition 3.1** (Directed Local Graph Transformation (DLGT))**.** *The* Directed Local Graph Transformation Problem *is defined as follows: Given two connected directed graphs $G_s, G_t$, find a computation of minimum length whose initial graph is $G_s$ and whose final graph is $G_t$.*

**Definition 3.2** (Undirected Local Graph Transformation (ULGT)**.** *The* Undirected Local Graph Transformation Problem *is defined as follows: Given two connected undirected graphs $G_s, G_t$, find a computation of minimum length whose initial graph is $G_s$ and whose final graph is $G_t$.*

Correspondingly, we define the following two decision problems:

**Definition 3.3** ($k$-Directed Local Graph Transformation (K-DLGT))**.** *The $k$-Directed Local Graph Transformation Problem is defined as follows: Given a positive integer $k$ and two connected directed graphs $G_s$ and $G_t$, decide whether there is a computation with initial graph $G_s$ and final graph $G_t$ of length at most $k$.*

**Definition 3.4** ($k$-Undirected Local Graph Transformation (K-ULGT))**.** *The $k$-Undirected Local Graph Transformation Problem is defined as follows: Given a positive integer $k$ and two connected undirected graphs $G_s$ and $G_t$, decide whether there is a computation with initial graph $G_s$ and final graph $G_t$ of length at most $k$.*

## 3.2. NP-hardness Results

In this section, we show the NP-hardness of the Undirected Local Graph Transformation Problem by proving the NP-hardness of K-ULGT (see Section 3.2.1). Since K-DLGT's NP-hardness can be proven very similar to K-ULGT's, we only briefly sketch the differences between the proofs in Section 3.2.2.

Throughout this section, for any positive integer $i$ we use the notation $[i]$ to refer to the set $\{1, 2, \ldots, i\}$.

### 3.2.1. k-ULGT Is NP-hard

We prove K-ULGT's hardness via a reduction from the Boolean satisfiability problem (SAT), which was proven to be NP-hard by Cook [Coo71] and, independently, by Levin [Lev73]. We briefly recap SAT as follows:

**Definition 3.5** (SAT)**.** *Given a set $X$ of $n$ Boolean variables $x_1, \ldots, x_n$ and a Boolean formula $\Phi$ over the variables in $X$ in conjunctive normal form (CNF), decide whether there is a truth assignment $t : X \to \{0, 1\}$ that satisfies $\Phi$.*

To reduce SAT to K-ULGT, we use the following reduction function:

**Definition 3.6** (Reduction function for $SAT \leq_p$ K-ULGT)**.** *Let $S = (X, \Phi)$ be a SAT instance, in which $X = \{x_1, \ldots, x_n\}$ is the set of Boolean variables and $\Phi = C_1 \wedge \cdots \wedge C_m$ for clauses $C_1, \ldots, C_m$. Then $f(S) = (G_s, G_t, \kappa)$ in which $\kappa = 2n + m$ and $G_s$ and $G_t$ are undirected graphs defined as follows. Without loss of generality, assume that each literal $y_i \in \{x_i, \overline{x_i}\}$ occurs only once in each clause.*

*We define the following two sets of nodes: $V_C = \{C_1, \ldots, C_m\}$ and $V_{X_i} = \{x_i, \overline{x_i}, s_i, t_i\}$. Then, the set of nodes of $G_s$ and $G_t$ is $V = \bigcup_{1 \leq i \leq n} V_{X_i} \cup V_C \cup \{r\}$. For the set of edges, define $E_{X_i} = \{\{s_i, x_i\}, \{s_i, \overline{x_i}\}, \{x_i, t_i\}, \{\overline{x_i}, t_i\}\}$ for every $i \in [n]$, $E_{C_j} = \{\{y_i, C_j\} | y_i \in \{x_i, \overline{x_i}\} \wedge y_i$ occurs in $C_j\}$ for every $j \in [m]$, $E_{sr} = \{\{s_i, r\} | 1 \leq i \leq n\}$, $E_{tr} = \{\{t_i, r\} | 1 \leq i \leq n\}$, $E_{Cr} = \{\{C_j, r\} | 1 \leq j \leq m\}$. Both $G_s$ and $G_t$ have the edges in $\bigcup_{1 \leq i \leq n} E_{X_i} \cup \bigcup_{1 \leq j \leq m} E_{C_j}$. Additionally, $G_s$ has the edges in $E_{sr}$ and $G_t$ has the edges in $E_{tr} \cup E_{Cr}$.*

Intuitively, each variable $x_i$ is mapped to a *gadget* $X_i$ consisting of the four nodes $x_i, \overline{x_i}, s_i$ and $t_i$. Also each clause $C_j$ is connected with each literal occurring

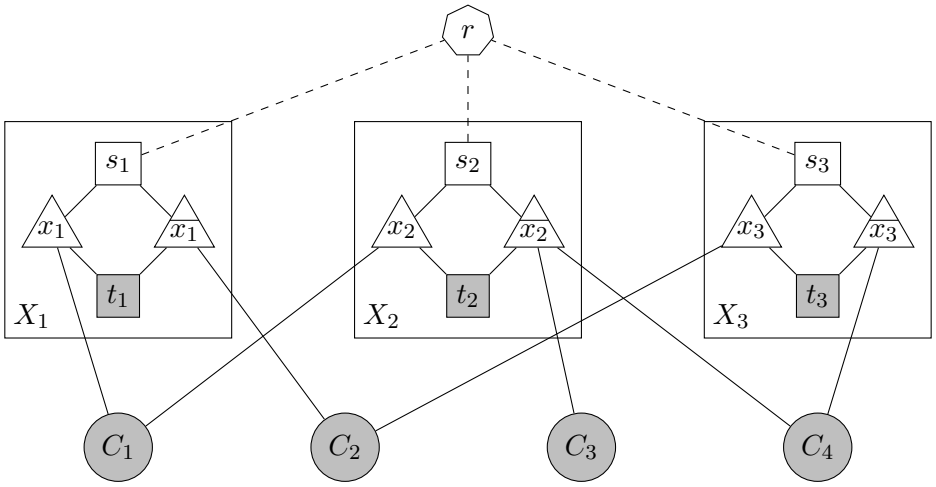



Figure 3.1.: Graph $G_s$ returned by the reduction function in the **undirected** case for the (example) Boolean formula $(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_2}) \wedge (\overline{x_2} \vee \overline{x_3})$. $G_t$ differs from $G_s$ in that the dashed edges do not exist and all grey nodes share an edge with node $r$.

within it. Lastly, in $G_s$, each of the $s_i$ is connected with the node $r$, whereas in $G_t$, each of the $t_i$ and each of the $C_j$ are connected with $r$. Figure 3.1 shows an example of the output of the reduction function for a given formula in CNF.

We now show that every SAT instance $S$ is satisfiable if and only if $f(S)$ is a "yes" instance of $\kappa$-ULGT. We start with the "only if" part because this is the simpler direction:

**Lemma 3.7.** *If a* SAT *instance $S$ as in Definition 3.6 is satisfiable then $f(S) = (G_s, G_t, \kappa)$ with $\kappa = 2n + m$ is a* $\kappa$-ULGT *instance and there is a computation with initial graph $G_s$ and final graph $G_t$ of a length of at most $2n + m$.*

*Proof.* Assume there is a satisfying truth assignment $t : X \to \{0, 1\}$ of $S$. For every $1 \leq i \leq n$ let $y_i := x_i$ if $t(x_i) = 1$ or $y_i := \overline{x_i}$ if $t(x_i) = 0$. We construct the following computation with initial graph $G_s$ and final graph $G_t$:

1. For every $1 \leq i \leq n$, $s_i$ delegates the edge $\{s_i, r\}$ to $y_i$.
2. For every $C_j \in \{C_1, \ldots, C_m\}$ choose one neighbor $z_j \in \{y_1, \ldots, y_n\}$ (we show below that this exists) and let $z_j$ introduce $r$ to $C_j$.
3. For every $1 \leq i \leq n$, $y_i$ delegates the edge $\{y_i, r\}$ to $t_i$.

Obviously, the length of this computation is $2n + m$. To prove the missing part, recall that every $C_j$ is satisfied under $t$. This means that there is at least one literal $z_j$ in $C_j$ that evaluates to true: i.e., there is an $i \in [n]$ such that $z_j = x_i$ if $t(x_i) = 1$ or $z_j = \overline{x_i}$ if $t(x_i) = 0$. By definition of $y_i$, $z_j = y_i$. Thus because $z_j$ occurs in $C_j$, $y_i$ was a neighbor of $C_j$ during Step 2. □

The "if" part is more complex. We begin with the following insight that will prove helpful in the course of this part.

**Lemma 3.8.** *Suppose the nodes in the initial graph of a computation $C$ can be decomposed into disjoint sets $V_1, \ldots, V_k, P$ such that there is no edge $\{u, v\}$ for any $u \in V_i$, $v \in V_j$, $i, j \in [k]$, $i \neq j$ and that throughout $C$ none of the nodes in $P$ applies a primitive. Then there is no edge $\{u, v\}$ for any $u \in V_i$, $v \in V_j$, $i, j \in [k]$, $i \neq j$ in any graph of the computation.*

*Proof.* Assume there is a computation $C$ and sets $V_1, \ldots, V_k, P$ as defined above and assume for contradiction that the claim is not true. We consider the first edge $\{u, v\}$ such that $u \in V_i$, $v \in V_j$, $i, j \in [k]$, $i \neq j$. Clearly, it cannot have been created by the application of a fusion primitive. Thus it must have been created by an introduction or delegation primitive applied by a node $w$ that knew both $u$ and $v$ before the application of this primitive. This implies $w \in P$. However, the nodes in $P$ do not apply any primitives by assumption, which yields a contradiction. □

In the rest of this section, we use the following notation: For a given instance of κ-ULGT $(G_s, G_t, k)$, we say a computation is *feasible* if and only if its initial graph is $G_s$, its final graph is $G_t$ and its length is at most $2n + m$. Furthermore, we say that the edge that is established during the application of an introduction or delegation primitive (the edge $(v, w)$ in Figure 1.1(a) and Figure 1.1(b)) is the *result* of the introduction or delegation, respectively.

The next lemma we show represents a main building block of the proof of the "if" part.

**Lemma 3.9.** *Let $S$ be a* SAT *instance and let $(G_s, G_t, k) = f(S)$. For every computation $C$ with initial graph $G_s$ and final graph $G_t$ of a length of at most $2n + m$ it holds: There are $y_1, \ldots, y_n$, $y_i \in \{x_i, \overline{x_i}\}$ for every $i \in [n]$, such that in $C$ there are no edges other than $E(G_s) \cup E(G_t) \cup \{\{y_i, r\} | i \in [n]\}$ and no edge occurs twice in any graph of $C$ (where $E(G_s)$ and $E(G_t)$ denote the edge sets of $G_s$ and $G_t$, respectively).*

The general idea of the proof of Lemma 3.9 is the following: To obtain the final graph, for each $j \in [m]$ the edge $\{C_j, r\}$ has to be created and for each $i \in [n]$ the edge $\{t_i, r\}$ has to be created. Each of these creations involves a distinct application of a primitive. Therefore, only $n$ applications of primitives are left in a feasible computation. We show that the nodes in each gadget $i$ have to apply at least one primitive $p_i$ that does not create one of the above edges. This implies that each gadget may apply no other primitive than $p_i$ to create an edge that is not in the final graph and that the nodes $r$ and $C_j$ themselves cannot apply any primitives at all, which by Lemma 3.8 means that there are no inter-gadget edges. We use these facts to prove that $p_i$ is used to remove the edge $\{s_i, r\}$ thereby creating either $\{x_i, r\}$ or $\{\overline{x_i}, r\}$.

We split the full proof of Lemma 3.9 into three claims, which we prove individually. The first claim we show is the following:

**Claim 3.10.** *In every feasible computation for every $i \in [n]$, there is a node $z_i \in \{s_i, x_i, \overline{x_i}\}$ that applies a fusion primitive or an introduction or delegation primitive whose result is not in $E_c := \{\{C_j, r\} | 1 \leq j \leq m\} \cup \{\{t_i, r\} | 1 \leq i \leq n\} \cup \{\{C_j, C_l\} | i, j \in [m]\} \cup \{\{t_i, C_j\} | i \in [n], j \in [m]\} \cup \{\{t_i, t_k\} | i, k \in [n]\}$.*

*Proof.* Assume for contradiction that there is a feasible computation and an $i \in [n]$ such that there is no $z_i \in \{s_i, x_i, \overline{x_i}\}$ that applies a fusion primitive or an introduction or delegation primitive whose result is not in $E_c$. Note that $s_i$ cannot delegate away any of its incident edges $\{s_i, x_i\}$ or $\{s_i, \overline{x_i}\}$ as the result would not be in $E_c$ (for it would be incident to $x_i$ or $\overline{x_i}$). Similarly, $x_i$ and $\overline{x_i}$ could not delegate away $\{x_i, s_i\}$ and $\{\overline{x_i}, s_i\}$, respectively. Therefore, the edges $\{x_i, s_i\}$ and $\{s_i, \overline{x_i}\}$ must be kept throughout the computation. Now observe that $s_i$ has an initial degree of three. Since $s_i$ has a degree of two in the final graph, there must be at least one application of a primitive in which $s_i$'s degree decreases. Consider the last such application: i.e., the resulting neighborhood of $s_i$ is $x_i$ and $\overline{x_i}$ (remember that these edges persist throughout the computation). If it was an application of a fusion primitive, then $x_i$, $s_i$ or $\overline{x_i}$ must have applied this primitive, yielding a contradiction. Otherwise, it must have been a delegation primitive applied by $s_i$ for no other primitive could then reduce $s_i$'s degree. However, the result of this delegation primitive must then be an edge incident to $x_i$ or $\overline{x_i}$, i.e., an edge not in $E_c$, yielding a contradiction in this case as well. All in all, we have proven Claim 3.10. □

The second claim we show for the proof of Lemma 3.9 is:

**Claim 3.11.** *For every feasible computation $C$, the following holds:*

*(i) for each $i \in [n]$ the nodes in $V_{X_i}$ may apply at most one primitive whose result is not an edge $\{t_k, r\}$ or $\{C_j, r\}$ for $k \in [n], j \in [m]$,*

*(ii) the nodes in $\{r\} \cup \{t_i | i \in [n]\} \cup \{C_j | j \in [m]\}$ do not apply any primitives at all, and*

*(iii) there is no graph in $C$ that contains an edge $\{u, v\}$ such that $u \in V_{X_i}$ and $v \in V_{X_k}$ for any $i, k \in [n]$ such that $i \neq k$.*

*Proof.* For the proof of this claim we distinguish between three types of edges. Edges $\{t_i, r\}$ for some $i \in [n]$ belong to type A. Edges $\{C_j, r\}$ for some $j \in [m]$ belong to type B. Last, every edge $e \notin E_c$ belongs to type $C$. Note that in order to obtain the final graph, the nodes need to establish $n$ edges of type A, $m$ edges of type B and, according to Claim 3.10, for every $i \in [n]$ one of the nodes in $\{s_i, x_i, \overline{x_i}\}$ has to apply a primitive whose result is an edge of type C or which is a fusion primitive. Since there may be at most $2n + m$ applications of primitives, no other primitives may be applied. This immediately proves (i). Since $r$ cannot create the first instance of an edge of type A or B by itself, $r$ cannot apply any primitive at all as this would require more than $2n + m$ primitive applications

($n + m$ are used to create the type A/B edges and additional $m$ primitives are applied by the nodes from $\{s_i, x_i, \overline{x_i} | i \in [n]\}$). For the same reason, the nodes in $\{t_i | i \in [n]\} \cup \{C_j | j \in [m]\}$ cannot apply any primitives which, together with the fact that $r$ cannot apply any primitives, equates to (ii). In addition this implies that the initial graph can be decomposed into $V_{X_1}, V_{X_2}, \ldots, V_{X_n}, P$ with $P = \{r\} \cup \{C_j | j \in [m]\}$ such that there is no edge $\{u, v\}$ for any $u \in V_{X_i}, v \in V_{X_k}$, $i, k \in [n]$ and $i \neq k$, and the nodes in $P$ do not apply any primitive at all. Thus, (iii) follows from Lemma 3.8. □

The last claim we show for the proof of Lemma 3.9 is the following:

**Claim 3.12.** *In every feasible computation, for every $i \in [n]$ there must be a graph containing an edge $\{x_i, r\}$ or $\{\overline{x_i}, r\}$.*

*Proof.* To prove Claim 3.12, we show that $s_i$ has to delegate $\{s_i, r\}$ to $x_i$ or $\overline{x_i}$. We do so by proving that $s_i$ cannot have a neighbor other than $x_i$, $\overline{x_i}$ or $r$, which is sufficient because $r$ does not apply any primitive according to (ii) of Claim 3.11. Assume for contradiction that $s_i$ has an edge $\{s_i, v\}$ such that $v \notin \{x_i, \overline{x_i}, r\}$ and let $\{s_i, v\}$ be the last such edge that occurs in a graph in the computation. Due to (iii) of Claim 3.11, $v \in \{t_i | i \in [n]\} \cup \{C_j | j \in [m]\}$. Consider the node $w$ that applied an introduction or delegation primitive whose result was the edge $\{s_i, v\}$. For this to be possible, there must have been an edge $\{w, s_i\}$ when $w$ applied the primitive. Since $r$, all $t_i$ for $i \in [n]$ and all $C_j$ for $j \in [m]$ do not apply any primitives according to (ii) of Claim 3.11 and because of (iii) of Claim 3.11, $w \in \{x_i, s_i, \overline{x_i}\}$. According to (i) of Claim 3.11, the computation may not contain another application (than this one) of a delegation / introduction primitive by a node in $V_{X_i}$ whose result is not an edge $\{t_k, r\}$ or $\{C_j, r\}$ for $k \in [n], j \in [m]$ (*). Since $\{s_i, v\} \notin E(G_t)$, this edge must be removed by some application of a delegation primitive. Note that since $v \in \{t_i | i \in [n]\} \cup \{C_j | j \in [m]\}$, $s_i$ must apply this primitive. Since $\{s_i, v\}$ is the last edge different from $\{s_i, r\}$, $\{s_i, x_i\}$ and $\{s_i, \overline{x_i}\}$, $v$ must be delegated to one of the nodes $r$, $x_i$ or $\overline{x_i}$. If $s_i$ delegates $\{s_i, v\}$ to $x_i$ or $\overline{x_i}$, then the result is an edge $\{x_i, v\}$ or $\{\overline{x_i}, v\}$, which contradicts (*). Thus assume $s_i$ delegates $\{s_i, v\}$ to $r$. Note that after this delegation, the edge $\{s_i, r\}$ exists and $s_i$ does not have a neighbor $v' \in \{t_i | i \in [n]\} \cup \{C_j | j \in [m]\}$ in any of the subsequent graphs (recall that $v$ was the last edge of its kind). Since $\{s_i, r\} \notin E(G_t)$ and $r$ does not apply any primitives according to (ii) of Claim 3.11, this edge must be delegated to either $x_i$ or $\overline{x_i}$ yielding an edge $\{x_i, r\}$ or $\{\overline{x_i}, r\}$, which contradicts (*) as well. As mentioned above this proves that $s_i$ has to delegate $\{s_i, r\}$ to $x_i$ or $\overline{x_i}$ and, as argued before as well, also implies the claim of the lemma. □

Claim 3.12 gives that during a feasible computation, the edges $\{t_i, r\}$ for all $i \in [n]$, $\{C_j, r\}$ for all $j \in [m]$ and $\{y_i, r\}$, $y_i \in \{x_i, \overline{x_i}\}$ for all $i \in [n]$ have to be created. Since each primitive can create at most one of these edges and the length of the computation is at most $2n + m$, this implies the claim of Lemma 3.9.

The rest of the proof of the "if" part, as formalized by the following lemma, is comparably straightforward.

**Lemma 3.13.** *Let $S$ be a* SAT *instance as in Definition 3.6. If $f(S) = (G_s, G_t, \kappa)$ with $\kappa = 2n + m$ is a* k-ULGT *instance and there is a computation with initial graph $G_s$ and final graph $G_t$ of a length of at most $2n + m$, then $S$ is satisfiable.*

*Proof.* Assume that $f(S) = (G_s, G_t, 2n + m)$ is a k-ULGT instance and there is a feasible computation $C$ for $f(S)$. According to Lemma 3.9 there are $y_1, \ldots, y_n$, $y_i \in \{x_i, \overline{x_i}\}$ for every $i \in [n]$ such that in $C$ there are no edges other than $E(G_s) \cup E(G_t) \cup \{\{y_i, r\} | i \in [n]\}$. Note that in $G_t$, for every $j \in [m]$ there is an edge $\{C_j, r\}$ and each such edge must have been the result of an introduction or delegation primitive applied by an $y_i$, $i \in [n]$ (as throughout $C$, the $C_j$ nodes do not have any other neighbors with an edge to $r$ that could possibly create this edge) . Let $g : \{C_1, C_2, \ldots, C_m\} \to \{y_1, y_2, \ldots, y_n\}$ be the mapping of each $C_j$ to the $y_i$ that applied a primitive that resulted in the edge $\{C_j, r\}$. Consider the truth assignment $t : X \to \{0, 1\}$ such that $t(x_i) = 1$ if $y_i = x_i$ and $t(x_i) = 0$ if $y_i = \overline{x_i}$. Observe that $t(y_i) = 1$ for every $i \in [n]$. Assume for contradiction that there is a clause $C_j$ in $S$ that does not evaluate to true under $t$. Note that $g(C_j)$ must occur in $C_j$ by construction. However, since $g(C_j) = y_i$ for some $i \in [n]$ and $t(y_i) = 1$, we obtain the desired contradiction. □

Putting both directions of the reduction's proof together, Lemma 3.7 and Lemma 3.13 imply SAT $\leq_p$ k-ULGT, from which we obtain:

**Corollary 3.14.** k-ULGT *is* NP*-hard.*

### 3.2.2. k-DLGT Is NP-hard

The proof of the NP-hardness of k-DLGT is very similar to that of k-ULGT. Therefore, we do not restate the whole proof but point out the differences between the two proofs.

The reduction function is a "directed version" of Definition 3.6, in which each of the edges is assigned a unique direction. Formally, it looks as follows:

**Definition 3.15** (Reduction function for $SAT \leq_p$ k-DLGT)**.** *Let $S = (X, \Phi)$ be a* SAT *instance, in which $X = \{x_1, \ldots, x_n\}$ is the set of Boolean variables and $\Phi = C_1 \wedge \cdots \wedge C_m$ for clauses $C_1, \ldots, C_m$. Then $f(S) = (G_s, G_t, \kappa)$ in which $\kappa = 2n + m$ and $G_s$ and $G_t$ are undirected graphs defined as follows. Without loss of generality, assume that each literal $y_i \in \{x_i, \overline{x_i}\}$ occurs only once in each clause.*

*We define the following two sets of nodes: $V_C = \{C_1, \ldots, C_m\}$ and $V_{X_i} = \{x_i, \overline{x_i}, s_i, t_i\}$. Then, the set of nodes of $G_s$ and $G_t$ is $V = \bigcup_{1 \leq i \leq n} V_{X_i} \cup V_C \cup \{r\}$. For the set of edges, define $E_{X_i} = \{(s_i, x_i), (s_i, \overline{x_i}), (x_i, t_i), (\overline{x_i}, t_i)\}$ for every $i \in [n]$, $E_{C_j} = \{(y_i, C_j) | y_i \in \{x_i, \overline{x_i}\} \wedge y_i$ occurs in $C_j\}$ for every $j \in [m]$, $E_{sr} = \{(s_i, r) | 1 \leq i \leq n\}$, $E_{tr} = \{(t_i, r) | 1 \leq i \leq n\}$, $E_{Cr} = \{(C_j, r) | 1 \leq j \leq m\}$. Both $G_s$ and $G_t$ have the edges in $\bigcup_{1 \leq i \leq n} E_{X_i} \cup \bigcup_{1 \leq j \leq m} E_{C_j}$. Additionally, $G_s$ has the edges in $E_{sr}$ and $G_t$ has the edges in $E_{tr} \cup E_{Cr}$.*

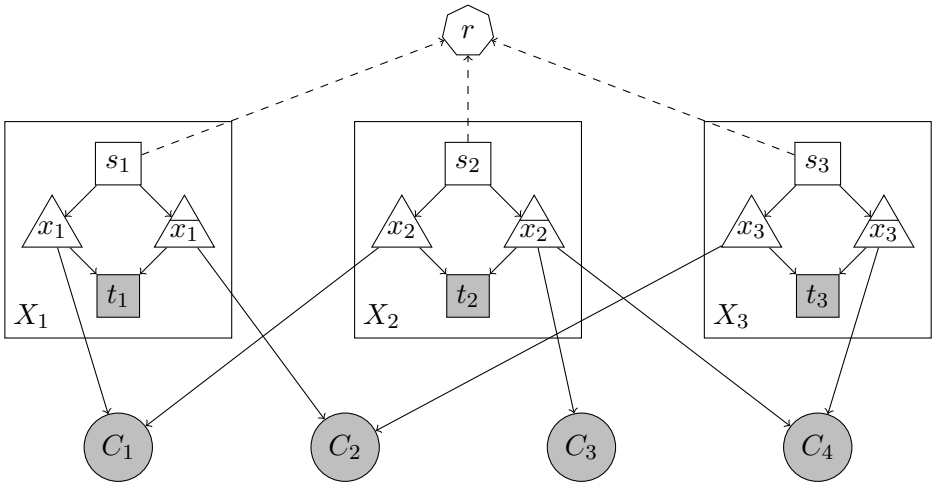



Figure 3.2.: Graph $G_s$ returned by the reduction function in the **directed** case for the (example) Boolean formula $(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_2}) \wedge (\overline{x_2} \vee \overline{x_3})$. $G_t$ differs from $G_s$ in that the dashed edges do not exist and all grey nodes share an outgoing edge with node $r$.

An example of the output of this reduction function is depicted in Figure 3.2. Note that this picture only differs from Figure 3.1 in that the edges are directed. More specifically, all solid edges are directed "downwards", the dashed edges are directed "upwards" and all grey nodes are supposed to have an "upward" edge to $r$ in $G_t$.

We now point out the differences in the proofs for the directed case, in which we refer to the lemmas from Section 3.2.1. Note that to shorten the description, every edge $\{u, v\}$ that appears in the proof for the undirected case should be read as the directed edge $(u, v)$ unless noted differently.

Lemma 3.7 (the "only if" part of the reduction) directly transfers to the directed case: the same approach described in that proof can be applied in the κ-DLGT scenario as well.

For the directed version of Lemma 3.8, the same proof applies, where the only additional argument to be mentioned is that the first edge $(u, v)$ such that $u \in V_i, v \in V_j, i, j \in [k], i \neq j$ not only cannot have been created by the application of a fusion primitive, but also not by the application of a reversal primitive.

For the proof of the counterpart of Lemma 3.9, we individually consider all three claims used for that lemma. The argument showing that the claim of the lemma follows from these three claims is analogous.

Claim 3.10 must be completed such that $z$ applied either a fusion primitive or an introduction, delegation or reversal primitive whose result is not in $E_c$. This set of edges is defined as: $E_c := \{(C_j, r) | 1 \leq j \leq m\} \cup \{(r, C_j) | 1 \leq j \leq m\} \cup \{(t_i, r) | 1 \leq i \leq n\} \cup \{(r, t_i) | 1 \leq i \leq n\} \cup \{(C_j, C_l) | i, j \in [m]\} \cup \{(t_i, C_j) | i \in$

$[n], j \in [m]\} \cup \{(C_j, t_i) | i \in [n], j \in [m]\} \cup \{(t_i, t_k) | i, k \in [n]\}$. In the proof, the assumption for contradiction includes that none of the $z_i$ applies a reversal primitive whose result is not in $E_c$. In addition to that $s_i$ cannot delegate away $(s_i, x_i)$ and $(s_i, \overline{x_i})$, we argue that $s_i$ cannot reverse this edge as the result would not be in $E_c$. These two facts immediately imply that $(s_i, x_i)$ and $(s_i, \overline{x_i})$ persist throughout the computation. For the last primitive application that reduces $s_i$'s degree to two, we have to take into account that this could also be the application of a reversal primitive. However, in that case, the result of this primitive application would be an edge whose one endpoint is $s_i$: i.e., an edge not in $E_c$, yielding a contradiction as well.

In the proof of Claim 3.11 we need the following additional argument to show that $r$ cannot apply any primitives (the previous argument that $r$ cannot create the first instance of an edge of type A or B by itself does not suffice if the reversal primitive could also be applied): Since no primitives other than the $n$ creating the type A edges, the $m$ creating the type $B$ edges and the $n$ creating the type $C$ edges or being a fusion primitive can be applied, there cannot be any edge $e \in E_c \setminus \{(t_i, r), (C_j, r) | i \in [n], j \in [m]\}$ in any graph of the computation. In particular, there can be no edge $(r, C_j)$ for any $j \in [m]$ or $(r, t_i)$ for any $i \in [n]$. Thus, $r$ cannot create the first instance of a type A or B edge via a reversal primitive and, together with the previous argument, we also obtain that $r$ cannot apply any primitive at all.

In the proof of Claim 3.12, in order to show that $s_i$ has to delegate $(s_i, r)$ to $x_i$ or $\overline{x_i}$, we similarly prove that $s_i$ cannot have an outgoing neighbor other than $x_i$, $\overline{x_i}$ or $r$. In the directed scenario, however, this is not immediately sufficient because $r$ does not apply any primitive. Instead, we additionally have to consider that $s_i$ cannot simply reverse edge $(s_i, r)$ as the resulting edge does not belong to $G_t$ and $r$ does not apply any primitive, which would be necessary to remove that edge again. After that, note that the edge $(s_i, v)$ such that $v \notin \{x_i, \overline{x_i}, r\}$, which we assume to exist for contradiction, cannot have been established by the application of a reversal primitive, for such a $v$ cannot apply any primitive at all according to (ii) of the adapted Claim 3.11 (since $v \in \{t_i | i \in [n]\} \cup \{C_j | j \in [m]\}$ follows equally in the directed case). Thus it is feasible to continue with the consideration of the $w$ that applied an introduction or delegation primitive whose result was $(s_i, v)$. The claim (*) in the directed case is that the computation may not contain another application (than the one applied by $w$ to create $(s_i, v)$) of a delegation, introduction or reversal primitive by a node in $V_{X_i}$ whose result is not an edge $(t_k, r)$ or $(C_j, r)$ for $k \in [n], j \in [m]$. Again, in order to remove $(s_i, v) \notin E(G_t)$, $s_i$ must apply a delegation primitive to remove this edge. In this case, the argument is that $s_i$ cannot reverse the edge, because $(v, s_i) \notin E(G_t)$ and $v$ does not apply any primitive (as argued before). In the last contradiction of the proof, which relies on that $(s_i, r) \notin E(G_t)$, there is not only the option that $s_i$ delegates this edge to $x_i$ or $\overline{x_i}$, but also that $s_i$ reverses this edge. This however would yield an edge $(r, s_i)$, which does not belong to $G_t$ and could not be removed, for $r$ does not apply any primitives.

The last lemma required is Lemma 3.13, but this is completely analogous. All in all, we obtain as for the undirected case:

**Corollary 3.16.** k-DLGT *is* NP*-hard.*

## 3.3. Approximation Algorithms

The main part of this section consists in describing and analyzing a constant approximation algorithm for ULGT (Section 3.3.1). As it turns out, an algorithm for DLGT can be obtained by a suitable adaptation of this algorithm, as we will elaborate on in Section 3.3.2.

As an ingredient our algorithms use a 2-approximation algorithm for the following problem:

**Definition 3.17** (Undirected Steiner Forest Problem (USF))**.** *The* Undirected Steiner Forest Problem *(*USF*) is defined as follows: For a given input* $(G, S)$ *such that* $G$ *is a graph and* $S$ *is a set of pairs of nodes from* $G$*, find a forest* $F$ *in* $G$ *with a minimum number of edges such that the two nodes of each pair in* $S$ *are connected by a path in* $F$*.*

Note that in the literature typically a more general definition of Steiner forest is used, in which edges may have arbitrary edge weights. However, when all these weights are set to one, we obtain the above problem and thus every algorithm for the more general definition can also be used for the problem as defined here. We will therefore use an existing 2-approximation algorithm (see Section 1.3.4 for an overview) as a black box without discussing it any further.

### 3.3.1. A Constant Approximation Algorithm for ULGT

We now describe an approximation algorithm for ULGT and prove it to have a constant approximation ratio.

#### Algorithm Description

For an initial graph $G_s = (V, E_s)$ and a final graph $G_t = (V, E_t)$, we define the set of *additional* edges $E_\oplus := E_t \setminus E_s$ and the set of *excess* edges $E_\ominus := E_s \setminus E_t$. We now describe the algorithm in detail and then summarize its pseudo-code in Algorithm 1. Our algorithm consists of two parts, the first of which deals with establishing all additional edges and the second of which deals with removing all excess edges.

In the first part, using an arbitrary 2-approximation algorithm for the USF as a black box, the algorithm computes a 2-approximate solution to the following USF instance: The given graph is $G_s$ and the set of pairs of nodes is $E_\oplus$. Note that the result is a forest such that for every edge $\{u, v\} \in E_\oplus$, $u$ and $v$ belong to the same tree. For each tree $T$ in this forest the algorithm then selects an arbitrary

root $r_T$ and connects all nodes in $T$ that are incident to an edge in $E_\oplus$ to $r_T$. The exact details of this will be described when we analyze the length of the resulting computation. In the next step, for every tree $T$ and every $\{u, v\} \in E_\oplus$ such that $u$ and $v$ belong to $T$, $r_T$ introduces $u$ and $v$ to each other, thereby creating the edge $\{u, v\}$. After that, the superfluous edges (i.e., those edges that belong neither to $G_s$ nor to $E_\oplus$) are deleted in a bottom-up fashion: Every node that does not have a descendant with a superfluous edge (in the tree $T$ that this node belongs to when viewing this tree as rooted by $r_T$), fuses all superfluous edges and delegates the last such edge to its parent in the tree. Note that all superfluous edges in the same tree $T$ have $r_T$ as one of their endpoints.

The second part of the algorithm is similar to the first, with the following differences: In the fifth step, the USF is approximated for the input $(G_t, E_\ominus)$. Note that the solution is a subgraph of the graph obtained after the first part of the algorithm. In the sixth step, only one of the two endpoints of an edge from $E_\ominus$ is selected to become connected with the root of the tree the endpoints belong to. In the seventh step (where in the first part the additional edges are created by the $r_T$ nodes), for each edge $e \in E_\ominus$, the endpoint selected in the sixth step delegates this edge to $r_T$ (resulting in the edge $\{r_T, v\}$). These edges can then be delegated and fused in a bottom-up fashion by the endpoints other than $r_T$ in Step 8. In contrast to Step 4, we begin with applying fusions here because the edges superfluous $\{r_T, v\}$ exist twice here (one was created in Step 6 and one in Step 7).

### Analysis

In this section, we show that Algorithm 1 is a constant approximation algorithm for ULGT, which proves the following theorem:

**Theorem 3.18.** *There is a constant-factor approximation algorithm for* ULGT.

For convenience we will analyze the two parts of the algorithm individually. Therefore, for a given initial graph $G_s$ and final graph $G_t$, let $ALG_1(G_s, G_t)$ be the length of the computation of the first part of the algorithm for this instance, $ALG_2(G_s, G_t)$ be the length of the computation of the second part and $ALG(G_s, G_t) := ALG_1(G_s, G_t) + ALG_2(G_s, G_t)$. Furthermore, let $OPT(G_s, G_t)$ be the length of an optimal solution to ULGT for initial graph $G_s$ and final graph $G_t$. We also define the intermediate graph $G' = (V, E_s \cup E_\oplus)$. In the course of the analysis we will establish a relationship between $ALG_1(G_s, G_t)$ and $OPT(G_s, G')$ and between $ALG_2(G_s, G_t)$ and $OPT(G', G_t)$. This will aid us in determining the approximation factor of Algorithm 1 due to the following lemma:

**Lemma 3.19.** $OPT(G_s, G') + OPT(G', G_t) \leq 2OPT(G_s, G_t) + |E_\oplus|$.

*Proof.* Let $\mathcal{P}$ denote the problem equal to ULGT with initial graph $G_s$ and final graph $G_t$ with the additional requirement that the computation must contain $G'$ and let $OPT'(G_s, G', G_t)$ be the length of an optimal solution to it.

**Algorithm 1** Approximation algorithm for ULGT

**Input:** Initial graph $G_s$ and final graph $G_t$.

*First part (add additional edges):*

1: Compute a 2-approximate solution $F_{ALG,\oplus}$ for the USF with input $(G_s, E_\oplus)$.
2: For each tree $T$ in $F_{ALG,\oplus}$, select a root node $r_T$ and connect all nodes in $T$ that are incident to an edge in $E_\oplus$ with $r_T$ (for details see the proof of Lemma 3.20).
3: For each $\{u, v\} \in E_\oplus$, the root of the tree that $u$ and $v$ belong to applies the introduction primitive to create the edge $\{u, v\}$.
4: For each tree $T$ in $F_{ALG,\oplus}$, delegate all superfluous edges (i.e., not belonging to $G_s$ or $E_\oplus$) created during Step 2 bottom up in $T$ rooted at $r_T$, starting with the lowest level. At each intermediate node fuse all of these edges before delegating them to the next parent.

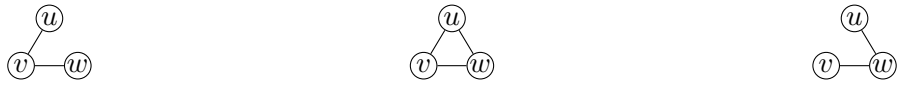
*Second part (remove excess edges):*

5: Compute a 2-approximate solution $F_{ALG,\ominus}$ for the USF with input $(G_t, E_\ominus)$.
6: For each $e \in E_\ominus$, let $s(e)$ be an arbitrary of the two endpoints of $e$. For each tree $T$ in $F_{ALG,\ominus}$, select a root node $r_T$ and for each $e \in E_\ominus$ whose endpoints belong to $T$, connect $s(e)$ with $r_T$ (similar to Step 2, for details see the proof of Lemma 3.21).
7: For each $e \in E_\ominus$, $s(e)$ delegates $e$ to $r_T$.
8: For each tree $T$ in $F_{ALG,\ominus}$, delegate all superfluous edges (i.e., not belonging to $G_t$) bottom-up while fusing multiple edges as in Step 4.

Clearly, $OPT(G_s, G') + OPT(G', G_t) = OPT'(G_s, G', G_t)$ (note that one could split the optimal solution to $\mathcal{P}$ at $G'$). We now show that $OPT'(G_s, G', G_t) \leq 2OPT(G_s, G_t) + |E_\oplus|$.

Consider a computation $C$ whose initial graph is $G_s$, whose final graph is $G_t$ and whose length is $OPT(G_s, G_t)$ (note that such a computation is an optimal solution to ULGT). We now transform $C$ into a computation that represents a solution to $\mathcal{P}$. This transformation increases its length by only $OPT(G_s, G_t) + |E_\oplus|$ and thus proves the above claim.

Note that every edge $\{u, v\} \in E_\ominus$ is removed during $C$. This may happen due to the application of a fusion primitive (in case $G_t$ contains another edge; recall that edges may have a multiplicity of more than one) or a delegation primitive. For all edges to which the former case applies, we simply omit that application of a fusion primitive. For all edges to which the latter case applies, we replace each of these applications of delegation primitives by applications of introduction primitives. Altogether, we obtain a new computation $C'$ of equal length. Note that changing these primitive applications does not make the computation infeasible as this only causes the graph to have additional edges. The final graph of $C'$ is $(V, E_t \cup E_\ominus) = (V, E_s \cup E_\oplus) = G'$ (recall that $E_t = (E_s \cup E_\oplus) \setminus E_\ominus$). Next we append $C'$ by $C$ and obtain the computation $C''$ of a length of $2OPT(G_s, G_t)$.




(a) Initial graph $G_s$. (b) Intermediate graph $G'$. (c) Final graph $G_t$.

Figure 3.3.: Example in which $OPT(G_s, G') + OPT(G', G_t) = 2OPT(G_s, G_t) + |E_\oplus|$: Transforming $G_s$ to $G'$ requires one application of an introduction primitive. To transform $G'$ into $G_t$, one needs to apply a delegation and a fusion primitive. $OPT(G_s, G_t)$, on the other hand, is one since $v$ can simply delegate $\{u, v\}$ to $w$.

Note that since $C$ transformed $G_s$ to $G_t$, this second half of $C''$, which starts from $G' = (V, E_s \cup E_\oplus)$, has the final graph $G'' = (V, E_t \cup E_\oplus)$: i.e., each edge from $E_\oplus$ appears twice in $G''$. Thus we extend $C''$ by fusing each edge from $E_\oplus$ with its double, resulting in a computation $C'''$ of a length of $2OPT(G_s, G_t) + |E_\oplus|$. Since $C'''$ represents a solution to $\mathcal{P}$ for initial graph $G_s$ and final graph $G_t$, this completes the proof. □

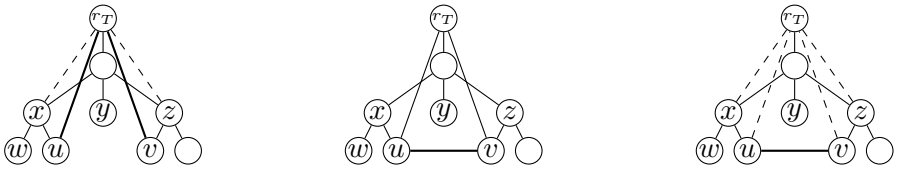
One might question whether this bound is tight or, for example, if it is possible to get rid of the additional $|E_\oplus|$ addend by a more careful analysis. Figure 3.3 shows that this bound is indeed tight and shows an example where $OPT(G_s, G') + OPT(G', G_t) = 2OPT(G_s, G_t) + |E_\oplus|$.

In the rest of the analysis we show that $ALG_1(G_s, G_t) \leq 11OPT(G_s, G')$ (Lemma 3.20) and that $ALG_2(G_s, G_t) \leq 11OPT(G', G_t)$ (Lemma 3.21). According to Lemma 3.19 this implies that $ALG(G_s, G_t) \leq 11(2OPT(G_s, G_t) + |E_\oplus|) \leq 33OPT(G_s, G_t)$ (since, clearly, $OPT(G_s, G_t) \geq |E_\oplus|$), which yields the claim of Theorem 3.18. We start with the former claim:

**Lemma 3.20.** $ALG_1(G_s, G_t) \leq 11OPT(G_s, G')$.

*Proof.* Let $F_{OPT,\oplus}$ be an optimal solution to the USF with input $(G_s, E_\oplus)$ and recall that $F_{ALG,\oplus}$ is the USF approximation computed in Step 1 of Algorithm 1. Throughout the analysis, $|F_{OPT,\oplus}|$ and $|F_{ALG,\oplus}|$ will denote the number of edges in these solutions. In the first part of this proof, we show that $ALG_1(G_s, G_t) \leq 4|F_{OPT,\oplus}| + 3|E_\oplus|$. The second part then consists in proving $OPT(G_s, G') \geq |F_{OPT,\oplus}| - |E_\oplus|$, which together with the observation that $OPT(G_s, G') \geq |E_\oplus|$ yields the claim.

To upper bound $ALG_1(G_s, G_t)$, we analyze the number of primitives applied in each of the steps of the first part of the approximation algorithm. In Step 1, no primitive is applied. To keep the number of edges as low as possible (which saves fusion primitives in Step 4), the algorithm for every $T$ in $F_{ALG,\oplus}$ connects the desired nodes to $r_T$ in Step 2 in the following way: To simplify the description, we view $T$ as rooted at $r_T$ and for a node $u \in T$ denote by $ST(u)$ the set consisting of $u$ and all of its descendants in the tree $T$ rooted at $r_T$. We say a node $u$ is




(a) Step 2 connects all endpoints of edges in $E_\oplus$ belonging to $T$ with $r_T$.

(b) In Step 3, $r_T$ creates the edges in $E_\oplus$ that belong to $T$ by an introduction.

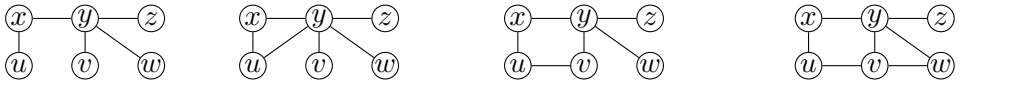
(c) Step 4 removes all superfluous edges by delegating and fusing them up in the tree.

Figure 3.4.: Example of a tree $T$ with root $r_T$ for Steps 2-4 of Algorithm 1 assuming $\{u, v\} \in E_\oplus$. $ST(x)$ consists of $x$, $w$ and $u$. $x$ is relevant, whereas $y$ is not. Dashed edges exist temporarily during the displayed step.

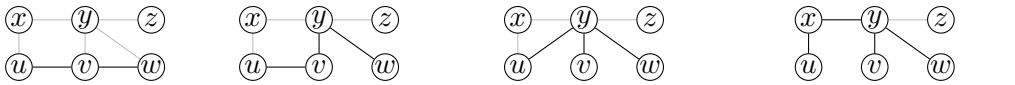
*relevant* if $ST(u)$ contains a node with an endpoint in $E_\oplus$[1]. See Figure 3.4 for an illustration of these notions. First of all, $r_T$ introduces itself to all relevant children. Then, starting from the second level, we proceed level-wise in the tree: For each level $i$, every node $u$ at level $i$ checks whether $u$ is an endpoint of an edge in $E_\oplus$. If so, $u$ introduces $r_T$ to all relevant children. Otherwise, $u$ introduces $r_T$ to all but one of its relevant children (chosen arbitrarily) and delegates $r_T$ to the relevant child it did not introduce $r_T$ to. The result of this procedure is that each node incident to an edge in $E_\oplus$ has an edge to $r_T$ for the tree $T$ it belongs to, see Figure 3.4(a). Note that according to the definition of $F_{ALG,\oplus}$, for each pair $\{u, v\} \in E_\oplus$, $u$ and $v$ belong to the same tree $T$. The above procedure increases the number of edges by at most $2|E_\oplus|$ and requires at most $|F_{ALG,\oplus}|$ applications of primitives (at most one for every edge in $T$). It is easy to see that Step 3 (c.f. Figure 3.4(b)) involves exactly $E_\oplus$ applications of primitives. For the length of Step 4 (c.f. Figure 3.4(c)), note that for every tree $T$ at most one delegation has to be applied for every edge in $T$ (causing $|F_{ALG,\oplus}|$ delegations in total) and at most $2|E_\oplus|$ fusions have to be applied, for this is the number of superfluous edges created during Step 2. All in all, Step 2, Step 3 and Step 4 involve $|F_{ALG,\oplus}|$, $|E_\oplus|$, and $|F_{ALG,\oplus}| + 2|E_\oplus|$ applications of primitives, respectively. This makes a total of $2|F_{ALG,\oplus}| + 3|E_\oplus|$. Since $F_{ALG,\oplus}$ is a 2-approximation of $F_{OPT,\oplus}$, we obtain $ALG_1(G_s, G_t) \leq 4|F_{OPT,\oplus}| + 3|E_\oplus|$.

For the lower bound on $OPT(G_s, G')$, assume for contradiction that there is a computation $C$ with initial graph $G_s$ and final graph $G'$ of a length of $L < |F_{OPT,\oplus}| - |E_\oplus|$. Let $G_s = G_1 \to G_2 \to \ldots \to G_L$ be the sequence of graphs of this computation. For every $\{u, v\} \in E_\oplus$ we iteratively create a path from $u$ to $v$ in the following way: Begin with $P^L_{u,v} := (u, v)$. Note that $P^L_{u,v}$ exists in $G_L$. We iterate through $C$ in reverse order and for every graph $G_i$, if $P^{i+1}_{u,v}$ exists in $G_i$, $P^i_{u,v} := P^{i+1}_{u,v}$. Otherwise, since $G_{i+1}$ is the result of a single application of a

[1]Note that although any tree in $F_{ALG,\oplus}$ that contains nodes that are not relevant for any root could trivially be reduced in size, we have to take into account that such trees exist since we treat the approximation algorithm for USF as a black box.



(a) Initial graph $G_1$. (b) $G_2$: $x$ has introduced $u$ to $y$. (c) $G_3$: $y$ has delegated $u$ to $v$. (d) $G_4$: $y$ has introduced $v$ to $w$.



(e) $G_4$: $P^4_{u,v} = (u, v)$, $P^4_{v,w} = (v, w)$. (f) $G_3$: $P^3_{u,v} = (u, v)$, $P^3_{v,w} = (v, y, w)$. (g) $G_2$: $P^2_{u,v} = (u, y, v)$, $P^2_{v,w} = (v, y, w)$. (h) $G_1$: $P^1_{u,v} = (u, x, y, v)$, $P^1_{v,w} = (v, y, w)$.

Figure 3.5.: Example of an optimal computation $C$ with initial graph $G_1$ and $E_\oplus = \{\{u, v\}, \{v, w\}\}$ and the notions used in the proof of Lemma 3.20. The upper row shows $C$ in order, the lower row illustrates the path sets $P^i_{u,v}$ and $P^i_{v,w}$, which are defined by iterating through $C$ in reverse order. In the lower row, the edges drawn in black in $G_i$ are the edges belonging to $F^i$. Observe that $F^1$ is a superset of a feasible solution to the USF for graph $G_1$ and node pairs $E_\oplus$.

primitive to $G_i$, there is exactly one edge $\{x, y\}$ in $P^{i+1}_{u,v}$ that exists in $G_{i+1}$ but not in $G_i$ and this edge was created by the application of an introduction or delegation primitive of some node $w$ such that $\{w, x\}$ and $\{w, y\}$ exist in $G_i$. Thus, let $P^i_{u,v}$ be $P^{i+1}_{u,v}$ with $(x, y)$ replaced by $(x, w, y)$ and note that $P^i_{u,v}$ exists in $G_i$. Eventually, we obtain a path $P^1_{u,v}$ that exists in $G_s$. For $i \in \{1, \dots, L\}$, let $F^i := \bigcup_{\{u,v\} \in E_\oplus} E(P^i_{u,v})$ (where $E(P)$ is the set of all edges on the path $P$) be a normal set of edges (i.e., not a multiset). Note that $F^1$ can be transformed into a feasible (though not necessarily optimal) solution to the USF with input $(G_s, E_\oplus)$ by removing cycle edges. Therefore, $|F^1| \geq |F_{OPT,\oplus}|$. An example is given in Figure 3.5. For an arbitrary $i \in \{1, \dots, L-1\}$, note that $|F^i| \leq |F^{i+1}| + 1$: if $G_{i+1}$ was obtained from $G_i$ by the application of a fusion primitive, this inequality trivially holds as none of the above paths changes in this case. Otherwise, $G_{i+1}$ was obtained from $G_i$ by an application of an introduction or delegation primitive by some node $w$ causing at most one edge $\{x, y\}$ to exist in $G_{i+1}$ that does not exist in $G_i$. In this case, we further know that $\{w, x\}$ and $\{w, y\}$ exist in $G_i$ and by the definition of the above paths, for every pair $\{u, v\}$ such that $P^{i+1}_{u,v}$ contains the edge $\{x, y\}$, the path $P^i_{u,v}$ contains $(x, w, y)$ as a sub-path instead and for all other pairs $\{u', v'\}$, $P^i_{u',v'} = P^{i+1}_{u',v'}$. By the definition of $F^i$ and $F^{i+1}$, this implies $|F^i| \leq |F^{i+1}| + 1$ also in this case. All in all, we obtain that $|F^1| \leq |F^L| + L = |E_\oplus| + L$ because $F^L = E_\oplus$ (note the definition of $F^L$). By the assumption that $L < |F_{OPT,\oplus}| - |E_\oplus|$, we obtain $|F^1| < |F_{OPT,\oplus}|$, which represents a contradiction. □

**Lemma 3.21.** *$ALG_2(G_s, G_t) \leq 11 OPT(G', G_t)$.*

*Proof.* The general structure of this proof follows the line of the proof of Lemma 3.20,

but differs in the details. Similar to the notation used in that proof, let $F_{OPT,\ominus}$ be an optimal solution for the USF with input $(G_t, E_\ominus)$ and recall that $F_{ALG,\ominus}$ is the USF approximation computed in Step 5 of Algorithm 1. Analogously, $|F_{OPT,\ominus}|$ and $|F_{ALG,\ominus}|$ denote the number of edges in these solutions. In the first part of this proof, we show that $ALG_2(G_s, G_t) \leq 4|F_{OPT,\ominus}| + 3|E_\ominus|$. The second part then consists in proving $OPT(G', G_t) \geq |F_{OPT,\ominus}| - |E_\ominus|$, which together with the observation that $OPT(G', G_t) \geq |E_\ominus|$ yields the claim.

To upper bound $ALG_2(G_s, G_t)$, we analyze the number of primitives applied in each step of the second part of the approximation algorithm. Of course, no primitive is applied in Step 5. The connections required in Step 6 can be created in a similar fashion as in Step 2, which is described in the proof of Lemma 3.20: For each tree $T$, we again proceed top-down in $T$ rooted at some arbitrary but fixed node $r_T$. Here, each intermediate node $u$ checks whether $u = s(e)$ for some $e \in E_\ominus$. If so, it introduces $r_T$ to all relevant children (here a node $v$ is *relevant* if $ST(v)$ contains a node $w$ such that $w = s(e')$ for some $e' \in E_\ominus$). Otherwise, it introduces $r_T$ to all but one relevant children and delegates it to the remaining one. In the end, for every edge $e \in E_\ominus$, $s(e)$ has an edge to $r_T$, the number of edges in the graph has increased by at most $|E_\ominus|$, and the process involved at most $|F_{ALG,\ominus}|$ applications of primitives. In Step 7, clearly exactly $|E_\ominus|$ edges have to be delegated. Step 8 is similar to Step 4 and for analogous reasons requires at most $|F_{ALG,\ominus}|$ delegations and at most $2|E_\ominus|$ fusions (recall that up to $|E_\ominus|$ edges were added in Step 6 and the edges delegated in Step 7 have to be removed as well). All in all, Step 6, Step 7 and Step 8 of the algorithm involve at most $|F_{ALG,\ominus}|$, $|E_\ominus|$ and $|F_{ALG,\ominus}| + 2|E_\ominus|$ applications of primitives, respectively, which yields: $ALG_2(G_s, G_t) \leq 2|F_{ALG,\ominus}| + 3|E_\ominus| \leq 4|F_{OPT,\ominus}| + 3|E_\ominus|$ (since $F_{ALG,\ominus}$ is a 2-approximation of $F_{OPT,\ominus}$).

To lower bound the value of $OPT(G', G_t)$, assume for contradiction that there is a computation $C$ with initial graph $G'$ and final graph $G_s$ of a length of $L < |F_{OPT,\ominus}| - |E_\ominus|$. Let $G_s = G_1 \to G_2 \to \ldots \to G_L$ be the sequence of graphs of this computation. Similar to the proof of Lemma 3.20, for every $\{u, v\} \in E_\ominus$, we create a path from $u$ to $v$, but this time we start with $P^1_{u,v} := (u, v)$ and consider the graphs in increasing order: For $i \in \{2, \ldots, L\}$, if $P^{i-1}_{u,v}$ exists in $G_i$, $P^i_{u,v} := P^{i-1}_{u,v}$. Otherwise since $G_i$ is the result of a single application of a primitive to $G_{i-1}$, there is exactly one edge $\{x, y\}$ in $P^{i-1}_{u,v}$ that exists in $G_{i-1}$ but not in $G_i$ and this edge must have been delegated by either $x$ or $y$ to some node $w$. In the following denote the node that applied the delegation by $z$ and denote by $\overline{z}$ the other node from $\{x, y\}$.In $G_{i-1}$, $z$ must share an edge with $w$ and this edge still exists in $G_i$ (for only one primitive is applied in the transition from $G_{i-1}$ to $G_i$). Since $\{z, \overline{z}\}$ was delegated by $z$ to $w$, the edge $\{w, \overline{z}\}$ exists in $G_i$. Thus, let $P^i_{u,v}$ be $P^{i-1}_{u,v}$ with $(x, y)$ replaced by $(x, w, y)$ and observe that $P^i_{u,v}$ exists in $G_i$. Eventually, we obtain a path $P^L_{u,v}$ that exists in $G_t$. Define $F^i := \bigcup_{\{u,v\} \in E_\ominus} E(P^i_{u,v})$ (where $E(P)$ is the set of all edges on the path $P$) as a normal set of edges (i.e., not a multiset) for every $i \in \{1, \ldots, L\}$, and note that $F^L$ can be transformed

into a feasible (though not necessarily optimal) solution to the USF with input $(G_t, E_\ominus)$ by removing cycle edges. Therefore, $|F^L| \geq |F_{OPT,\ominus}|$. Furthermore, for an arbitrary $i \in \{1, \ldots, L-1\}$, note that $|F^{i+1}| \leq |F^i| + 1$ because there is at most one edge $\{x, y\}$ that exists in $G_i$ but not in $G_{i+1}$ and thus causes the replacement of $(x, y)$ by $(x, w, y)$ for some fixed node $w$ for all paths that contain $(x, y)$ as a sub-path. This yields that $|F^L| \leq |F^1| + L = |E_\ominus| + L$ because $F^1 = E_\ominus$ (note the definition of $F^1$). By the assumption that $L < |F_{OPT,\ominus}| - |E_\ominus|$, we obtain $|F^L| < |F_{OPT,\ominus}|$, which represents a contradiction. □

As argued before, Lemma 3.20 and Lemma 3.21 (together with Lemma 3.19) imply Theorem 3.18.

### 3.3.2. A Constant Approximation Algorithm for DLGT

In this subsection we describe how to adapt Algorithm 1 to obtain a constant approximation algorithm for DLGT. The pseudocode of the adapted version of the algorithm is given in Algorithm 2 (with differences to Algorithm 1 highlighted in boldface). In this pseudocode and in the following proof, for a graph $G = (V, E)$, $U(G)$ denotes the undirected version of $G$: i.e., each edge $(u, v)$ in $G$ is replaced by $\{u, v\}$. An example of the procedure of Algorithm 2 is illustrated in Figure 3.6.

Algorithm 2 also has a constant approximation factor, which is stated by the following theorem:

**Theorem 3.22.** *There is a constant-factor approximation algorithm for* DLGT.

*Proof.* We begin with establishing a relationship between solutions to DLGT and to ULGT. For arbitrary directed graphs $G_s$ and $G_t$, let $C_d(G_s, G_t)$ be an optimal solution to DLGT with (directed) initial graph $G_s$ and (directed) final graph $G_t$ (this problem we denote by $P_d(G_s, G_t)$) and let $OPT_d(G_s, G_t)$ be its length. Let $C_u((U(G_s), U(G_t))$ be an optimal solution to ULGT with initial graph $U(G_s)$ and final graph $U(G_t)$, denote its length by $OPT_u((U(G_s), U(G_t))$ and denote this problem by $P_u((U(G_s), U(G_t))$. Let the computation $C'_u$ be obtained from $C_d$ by ignoring all edge directions and removing all applications of reversal primitives from $C_d$. Note that $C'_u$ is a solution to $P_u((U(G_s), U(G_t))$ and that its length is thus lower bounded by $OPT_u((U(G_s), U(G_t))$ and upper bounded by $OPT_d(G_s, G_t)$ (since we only shortened the optimal solution to $P_d(G_s, G_t)$). Therefore, $OPT_u((U(G_s), U(G_t)) \leq OPT_d(G_s, G_t)$ for all directed graphs $G_s$ and $G_t$. We will use this insight to compare the length of the solution computed by Algorithm 2 for initial graph $G_s$ and final graph $G_t$ with the length of the solution computed by Algorithm 1 for initial graph $U(G_s)$ and final graph $U(G_t)$ in order to prove the claim.

In the following, due to the similarities of Algorithm 2 with Algorithm 1, we only describe the changes to the algorithm that require additional explanation (namely, Step 2 and Step 6). The other changes (highlighted in boldface in Algorithm 2) are self-explanatory. Additionally, we analyze the influence of the differences in

**Algorithm 2** Approximation algorithm for DLGT

**Input:** Initial graph $G_s$ and final graph $G_t$.

*First part (add additional edges):*

1: Compute a 2-approximate solution $F_{ALG,\oplus}$ for the USF with input $(\mathbf{U}(\mathbf{G_s}), \mathbf{U}(\mathbf{E}_\oplus))$.

2: For each tree $T$ in $F_{ALG,\oplus}$, select a root node $r_T$, **reverse all edges in T that are oriented towards $\mathbf{r_T}$** and for all nodes $u$ in $T$ that are incident to an edge in $E_\oplus$, establish the edge $(\mathbf{r_T}, \mathbf{u})$ (for details see the proof of Theorem 3.22).

3: For each $(\mathbf{u}, \mathbf{v}) \in \mathbf{E}_\oplus$, the root of the tree $u$ and $v$ belong to applies the introduction primitive to create the edge $(\mathbf{u}, \mathbf{v})$.

4: For each tree $T$ in $F_{ALG,\oplus}$, **reverse all edges in T**, **reverse all superfluous edges (i.e., not belonging to $G_s$ or $E_\oplus$) created during Step 2** and delegate them bottom up in $T$ rooted at $r_T$, starting with the lowest level. At each intermediate node fuse all of these edges before delegating them to the next parent. **Afterwards, reverse all edges in T that were originally oriented away from $\mathbf{r_T}$ (i.e., the edges in T that were not reversed in Step 2).**

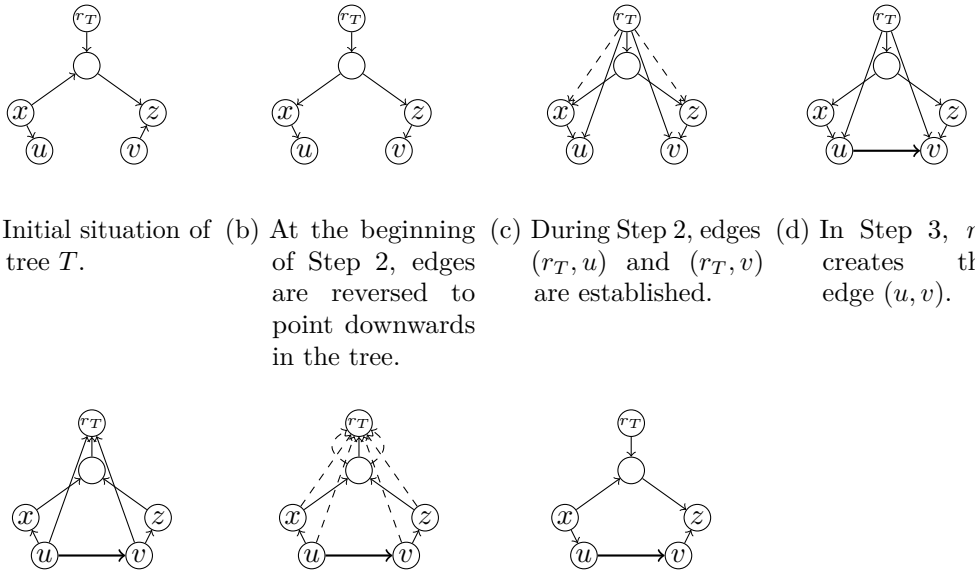
*Second part (remove excess edges):*

5: Compute a 2-approximate solution $F_{ALG,\ominus}$ for the USF with input $(\mathbf{U}(\mathbf{G_t}), \mathbf{U}(\mathbf{E}_\ominus))$.

6: For each tree $T$ in $F_{ALG,\ominus}$, select a root node $r_T$, **reverse all edges in T that are oriented towards $\mathbf{r_T}$** and for each $(\mathbf{u}, \mathbf{v}) \in \mathbf{E}_\ominus$ whose endpoints belong to $T$, establish the edge $(\mathbf{u}, \mathbf{r_T})$ (similar to Step 2, for details see the proof of Theorem 3.22).

7: For each $(\mathbf{u}, \mathbf{v}) \in \mathbf{E}_\ominus$, $\mathbf{u}$ delegates $(\mathbf{u}, \mathbf{v})$ to $r_T$.

8: For each tree $T$ in $F_{ALG,\ominus}$, **reverse all edges in T**, **reverse all edges created during Step 7** and delegate all superfluous edges (i.e., not belonging to $G_t$) bottom-up while fusing multiple edges as in Step 4. **Afterwards, reverse all edges in T that were originally oriented away from $\mathbf{r_T}$ (i.e., the edges in T that were not reversed in Step 6).**




(a) Initial situation of tree $T$.

(b) At the beginning of Step 2, edges are reversed to point downwards in the tree.

(c) During Step 2, edges $(r_T, u)$ and $(r_T, v)$ are established.

(d) In Step 3, $r_T$ creates the edge $(u, v)$.

(e) At the beginning of Step 4, all superfluous and all tree edges are reversed.

(f) During Step 4, the superfluous edges are delegated upwards in $T$.

(g) At the end of Step 4, the original edge orientations in $T$ are restored.

Figure 3.6.: Example of the first part of Algorithm 2 for a tree $T$ with root $r_T$ assuming $(u, v) \in E_\oplus$. Dashed edges exist temporarily during the displayed step.

the algorithms on the length of the computations in order to obtain the desired approximation factor.

The procedure for the creation of edges in Step 2 is, in general, very similar to that used in Algorithm 1 (described in the proofs of Lemma 3.20). There are three differences, though: First, in order to delegate / introduce edges downwards in a tree $T$, all edges of $T$ must be oriented downwards, which is why we first reverse all edges oriented into the opposite direction (towards $r_T$). Let $E_R$ be the set of all these edges of all trees $T$. Then this additionally involves $|E_R|$ applications of the reversal primitive in total. Second, since we deal with directed edges, we have to clarify that the edges delegated / introduced through the tree must be directed towards the root node. Third, this process creates the additional edges $(u, r_T)$ for every $u$ incident to an edge in $E_\oplus$, whereas $(r_T, u)$ is desired. Thus, for each such edge, we apply the reversal primitive to turn $(r_T, u)$ into $(u, r_T)$, which requires at most $2|E_\oplus|$ applications of primitives. In Step 4, we additionally have to reverse all edges in every tree first in order to be able to delegate upwards in the tree. This involves $|F_{ALG,\oplus}|$ additional applications of reversal primitives in total. Furthermore, we have to reverse all superfluous edges before we can start the bottom-up procedure. As their number is, again, upper bounded by $2|E_\oplus|$, we have

to apply only that number of reversal primitives. In addition, we afterwards reverse all edges whose orientation does not yet match their orientation in $G_s$, which totally requires an additional $|F_{ALG,\oplus} \setminus E_R|$ applications of reversal primitives (since these are exactly the edges in $F_{ALG,\oplus}$ that were not reversed in Step 2). Altogether, the additional number of primitives required in comparison to Algorithm 1 is $|E_R| + |2E_\oplus| + |F_{ALG,\oplus}| + 2|E_\oplus| + |F_{ALG,\oplus} \setminus E_R| = 2|F_{ALG,\oplus}| + 4|E_\oplus|$ Thus, the length $ALG_1^2(G_s, G_t)$ of the computation computed in the first part by Algorithm 2 is at most $ALG_1^1(U(G_s), U(G_t))+4|E_\oplus|+2|F_{ALG,\oplus}|$ (in which $ALG_1^1(U(G_s), U(G_t))$ is the length of the computation of the first part of Algorithm 1 for initial graph $U(G_s)$ and final graph $U(G_t)$).

For the second part, note that the creation of edges in Step 6 is generally very similar to that used in Algorithm 1 (described in the proof of Lemma 3.21). Here, for each edge $(u, v) \in E_\ominus$, $u$ corresponds to $s(e)$ in Algorithm 1. As in Step 2, we first have to reverse the edges downwards in the tree, incurring $r$ additional primitive applications. Again, the edges delegated / introduce through the tree must be directed towards the root node. In Step 8, we again have to reverse all edges in each tree, requiring an additional number of primitive applications of $|F_{ALG,\ominus}|$. Additionally, the edges created in Step 7 need to be reversed, which are $|E_\ominus|$ in total. Last, the edges whose orientation does not match their orientation in $G_t$ (which are those that were not reversed in Step 6) need to be reversed, which requires $|F_{ALG,\ominus}| - r$ applications of reversal primitives. All in all, the length $ALG_2^2(G_s, G_t)$ of the computation computed in the second part by Algorithm 2 is at most $ALG_2^1(G_s, G_t) + 2|F_{ALG,\ominus}| + |E_\ominus|$ (in which $ALG_2^1(U(G_s), U(G_t))$ is the length of the computation of the second part of Algorithm 1 for initial graph $U(G_s)$ and final graph $U(G_t)$).

Incorporating the arguments and results of the proofs of Lemma 3.20 and Lemma 3.21, we obtain:

$$\begin{aligned}
ALG_1^2(G_s, G_t) &\leq ALG_1^1(U(G_s), U(G_t)) + 4|F_{OPT,\oplus}| + 4|E_\oplus| \leq 8|F_{OPT,\oplus}| + 7|E_\oplus| \\
&\leq 8OPT(U(G_s), U(G')) + 15|E_\oplus| \leq 23OPT(U(G_s), U(G')), \text{ and} \\
ALG_2^2(G_s, G_t) &\leq ALG_2^1(U(G_s), U(G_t)) + 4|F_{OPT,\ominus}| + |E_\ominus| \leq 8|F_{OPT,\ominus}| + 4|E_\ominus| \\
&\leq 8OPT(U(G'), U(G_t)) + 12|E_\ominus| \leq 20OPT(U(G'), U(G_t)).
\end{aligned}$$

where $OPT$ is defined as in Section 3.3.1 and $G' = U((V, E_s \cup E_\oplus))$ for $E_s$ being the edge set of $G_s$.

Let $ALG^2(G_s, G_t)$ be the length of the computation computed by Algorithm 2 for initial graph $G_s$ and final graph $G_t$. Building on the above two inequations, we can apply Lemma 3.19 to obtain

$$\begin{aligned}
ALG^2(G_s, G_t) :=& ALG_1^2(G_s, G_t) + ALG_2^2(G_s, G_t) \\
&\leq 23(2OPT_u(U(G_s), U(G_t)) + |E_\oplus|) \\
&\leq 69OPT_u(U(G_s), U(G_t)) \leq 69OPT_d(G_s, G_t).
\end{aligned}$$

This finishes the proof. □

PART

# II

# Monotonic Searchability in Self-Stabilizing Topologies

The second main part of this thesis considers monotonic searchability and answers the following two questions: *How can monotonic searchability be obtained generally for a wide range of topologies? How can monotonic searchability be guaranteed when nodes can additionally leave the system?*
To answer the first question, we provide a general approach for monotonic searchability in graphs that contain the line graph as a subgraph. For a large number of graphs, our solution permits answering successful searches efficiently in the stabilized topology. To answer the second question, we provide a protocol that solves the finite departure problem together with monotonic searchability. For simplicity, we focus on line graphs for that solution.

**Outline of This Part** In Chapter 4 we describe a general framework to solve monotonic searchability in supergraphs of the line. In Chapter 5 we consider the combination of the $\mathcal{FDP}$ with monotonic searchability for the line topology.

CHAPTER 4

# Monotonic Searchability for Supergraphs of the Line

In this chapter, we provide a framework for monotonic searchability that is not restricted to a single class of graphs but can be applied to a wide range of topologies: line-supergraphs. The key idea to enable monotonic searchability in such a large class of graphs is to restrict the manipulation of edges to a suitable set of graph transformation primitives. These graph transformation primitives are based on the primitives in $\mathcal{IDF}$. Our framework is general enough to allow existing protocols to be transformed such that they satisfy monotonic searchability, which we illustrate with several examples. Beyond our general approach, at the end of this chapter we also provide a specific solution to monotonic searchability for the $Bridge{-}SKIP^{+}$ graph.

The main results of this chapter have previously appeared in the following publication:

> Christian Scheideler, Alexander Setzer, and Thim Strothmann. **Towards a Universal Approach for Monotonic Searchability in Self-stabilizing Overlay Networks**. In: *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*. Paris, France, 2016. [SSS16]

**Outline of This Chapter** We start this chapter in Section 4.1 with a definition of the communication model used in this part of the thesis and a formal statement of the problem considered here. After that, we introduce three new graph transformation primitives in Section 4.2 that one should use in a protocol to enable monotonic searchability. Following this, we show that conventional protocols for the self-stabilization of a topology can be transformed into ones using the new primitives in Section 4.3. Then, in Section 4.4, we present a generic search protocol according to which such protocols satisfy monotonic searchability and prove its correctness. After that, we give examples in Section 4.5 on how to apply our results to existing topologies — namely the list, the $SKIP^{+}$ graph and the linearized De Bruijn network. Last, as a small digression, in Section 4.6 we consider monotonic searchability for a particular topology called the $Bridge{-}SKIP^{+}$ *graph*, which is a supergraph of the $SKIP^{+}$ graph.

## 4.1. Communication Model and Problem Statement

The description of the system model in Section 2.1.2 deliberately left out a definition of the communication model as this differs in Part II and Part III of this thesis. In Section 4.1.1, we thus define the communication model for this part. After that, in Section 4.1.2 we formally state the problem considered in this chapter.

### 4.1.1. Communication Model

We assume that each node has a unique reference. For each node, there is a system-based variable called *channel* whose values are sets of messages. We denote the channel of a node $u$ as $u.Ch$ and it contains all incoming messages to $u$. Its message capacity is unbounded, although we assume that in each state there is only a finite number of messages in each channel. Furthermore, we assume that messages never get lost. A node can add a message to $u.Ch$ if it has a reference of $u$. Besides these channels there are no further communication means, so only point-to-point communication is possible.

In this model, a message $m$ requests to call an action $A$ at some node $u$ if $m$ corresponds to $A$ and $m$ is in the channel of $u$. When $A$ is executed because it was enabled due to $m$, $m$ is processed (in which $m$ is removed from $u.Ch$). Receiving and processing a message is considered an atomic step.

We assume *fair message receipt*. This means that if a computation contains a state in which there is a message in a channel of a node that enables an action in that node, then that action is eventually executed with the parameters of that message: i.e., the message is eventually processed.

The overlay network of a set of nodes is determined by their knowledge of each other. We say that there is a (directed) *edge* from $a$ to $b$, denoted by $(a, b)$, if node $a$ stores a reference of $b$ in its local memory or has a message in $a.Ch$ carrying the reference of $b$. In the former case, the edge is called *explicit* and in the latter case, the edge is called *implicit*. Messages can only be sent via explicit edges. Note that message receipt converts an implicit edge to an explicit edge since the message is in the local memory of a node while it is processed. With $NG$ we denote the directed *network (multi-)graph* given by the explicit and implicit edges. $ENG$ is the subgraph of $NG$ induced by only the explicit edges. We generally refer to the node set of the network graph by $V$.

We consider protocols that do not manipulate the internals of node references. Specifically, a protocol is *compare-store-send* if the only operations that it executes on node references are comparing them, storing them in a variable and sending them in a message. That is, operations on references such as addition, radix computation, hashing and similar are not used. In a compare-store-send protocol, a node may learn a new reference of a node only by receiving it in a message. A compare-store-send protocol cannot create new references. It can only operate on the references given to it.

It was proven in [NNS13] that if a compare-store-send protocol starts from a

disconnected network graph, the network graph remains disconnected in every state of the computation. Additionally, regarding the ability to form arbitrary topologies, [NNS13] proved that we need to assume that the initial state does not contain any nonexistent node references. Thus, throughout this chapter, for every computation we consider, we assume that initially the network graph is weakly connected and that for every reference stored in some node or some message, a node with that reference exists in the system. This assumption is common in the literature regarding self-stabilizing topologies (see, e.g., [BGP13; NNT13; KSS17]).

### 4.1.2. Problem Statement

The goal of this chapter is to provide a general solution for searching in a wide range of topologies that fulfill certain requirements. On a high level, these requirements are that the topologies are supergraphs of the line topology and that there exist self-stabilizing protocols for them that "monotonically improve" the nodes' outgoing edges by letting them get "closer" to their target. Apart from being correct, our solution allows to search efficiently once the topology has stabilized. To state the problem formally and completely, we now introduce a set of definitions.

First of all, we have to note that compare-store-send protocols are generally unable to satisfy monotonic searchability when starting from arbitrarily corrupted states. To formalize this, we briefly introduce the following notions: A *message invariant* $I$ is a predicate defined for a specific message type and over the parameters of the message type and the system state. For every protocol a number of message invariants may be specified. A message $m$ is called *corrupted* if $m$ is in the channel of a node and violates at least one message invariant. A state $S$ is called *admissible* if there are no corrupted messages in $S$. We say a (self-stabilizing) protocol *unconditionally satisfies monotonic searchability* if it satisfies monotonic searchability starting from any state. In [SSS15] it was proven that no self-stabilizing compare-store-send protocol can unconditionally satisfy monotonic searchability (Lemma 1 of [SSS15]). Consequently, to prove monotonic searchability for a protocol (according to a given search protocol $P_S$) it is sufficient to show that: (i) in every computation of the protocol every state subsequent to an admissible state is admissible as well, (ii) in every computation of the protocol there is an admissible state, and (iii) the protocol satisfies monotonic searchability according to $P_S$ in every computation that starts from an admissible state. Note that we have not defined any invariants yet and it is possible to pick invariants such that the set of admissible states equals the set of legitimate states in which the problem becomes trivial. However, for the invariants we provide, any initial topology can exist in an admissible state. In particular, as long as no corrupted messages are initially in the system, our protocols satisfy monotonic searchability throughout the computation.

To capture the class of topologies our generic solution can be used for, we introduce the notion of a *feasible topology*:

**Definition 4.1** (Feasible Topology)**.** *A topology $T$ is called* feasible *if and only if:*

1. *there is a total order* $\leq$ *on the identifiers of the nodes, and*
2. *for each pair of nodes* $u$ *and* $v$*, there is a path* $Q = (u = x_1, x_2, \ldots, x_k = v)$ *from* $u$ *to* $v$ *such that* $id(x_i) < id(x_{i+1})$ *for all* $1 \leq i < k$ *or* $id(x_{i+1}) < id(x_i)$ *for all* $1 \leq i < k$*.*

This definition is equivalent to that the topology is a supergraph of the line topology. But this representation will turn out helpful later on. Due to the total order defined on the node identifiers, we introduce the following notation as well: for three nodes $u, v, w$ we say $v$ is *closer* to $u$ than $w$ if and only if $|id(v) - id(u)| < |id(w) - id(u)|$.

For each feasible topology $T$, we assume the existence of a search protocol $S_T$ for $T$, called the *fast search protocol*. The name is chosen due to the assumption that the fast search protocol exploits the topology to find other nodes fast, although this is not a requirement. Formally, the fast search protocol needs to fulfill the following property:

**Definition 4.2** (Fast Search Protocol, Search Edges, Non-Search Edges)**.** *For a feasible topology* $T$*, a protocol* $S_T$ *is called a* fast search protocol for $T$ *if and only if:*

1. *for every pair of nodes source and dest in* $T$*,* SEARCH*(source,* $id(dest)$*) always succeeds, and*
2. *for every* SEARCH*(source,* $id(dest)$*) message, every node* $u$ *only forwards the search message to a node* $v$ *such that:* $\min(id(u), id(dest)) \leq id(v) \leq \max(id(u), id(dest))$*.*

*The set of edges in* $T$ *can be partitioned such that every edge that may possibly be used in* $S_T$ *is called a* search edge*, whereas every edge that is never used in* $S_T$ *is called a* non-search edge*.*

Intuitively, every SEARCH$(source, id(dest))$ message is always forwarded into the same "direction" (w.r.t. the total order of the identifiers) and never oversteps *dest*. The notion of (non-)search edges is introduced to allow for a wider range of topologies. As we will see, non-search edges do not need to be treated as carefully as search edges during self-stabilization to allow for monotonic searchability.

Throughout this chapter, we assume that the references for search and non-search edges are stored in distinct variables in the stabilization protocols. We make use of the following notions:

**Definition 4.3** ((Non-) Search Variables, Search Neighborhood)**.** *For a topology* $T$ *and a fast search protocol* $S_T$ *for* $T$*, let* $P$ *be a protocol for the self-stabilization of* $T$*. Each variable of* $P$ *is either a* search variable*, meaning that the references stored in this variable are used by* $S_T$ *or a* non-search variable *otherwise. In the legal states of* $P$*, all search edges of* $T$ *must be stored in search variables.*

*Accordingly, the* search neighborhood $\Gamma^*(u)$ *of a node* $u$ *consists of all nodes whose references are stored in search variables of* $u$*.*

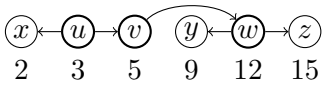



Figure 4.1.: Example graph with the identifiers listed below the nodes. Each node is drawn thick if and only if it belongs to $R(u, 12)$.

We will make use of the following notion of a *straight path*, which denotes a path of search edges such that identifiers of the nodes on the path are strictly monotonically increasing or decreasing:

**Definition 4.4** (Straight Path)**.** *A path $P = (x_1, x_2, \ldots, x_k)$ from $x_1$ to $x_k$ is called* straight *if and only if:*

1. *$(x_i, x_{i+1})$ is an explicit search edge for every $i \in \{1, \ldots, k-1\}$, and*
2. *either $id(x_1) < id(x_i) < id(x_k)$ or $id(x_k) < id(x_i) < id(x_1)$ for every $i \in \{2, \ldots, k-1\}$.*

For convenience we also define the following two functions:

**Definition 4.5** ($R(u, ID)$ / $R(U, ID)$)**.** *For a node $u$ and an identifier $ID$, $R(u, ID)$ is the set containing all nodes $v$ such that (i) $\min(id(u), ID) \leq id(v) \leq \max(id(u), ID)$, and (ii) there is a (possibly empty) straight path from $u$ to $v$. Furthermore, for a set $U$, $R(U, ID) := \bigcup_{u \in U} R(u, ID)$.*

Note that since the straight path may be empty, by this definition $u \in R(u, ID)$ for every node $u$ and identifier $ID$. An example of a set $R(u, ID)$ is illustrated in Figure 4.1.

We will show that a broad class of existing self-stabilizing protocols can be transformed to satisfy monotonic searchability. More specifically, we will consider protocols that fulfill the *monotonic convergence property (*MCP*)* defined as follows:

**Definition 4.6** (Monotonic Convergence Property (MCP))**.** *For a feasible topology $T$ and a fast search protocol $S_T$ for $T$, a protocol $P$ fulfills the* monotonic convergence property *for $T$ w.r.t. $S_T$ if and only if:*

1. *for every node $u$ and every search variable $Var$ there are two functions $\gamma^{min}_{u,Var}(A)$ and $\gamma^{max}_{u,Var}(A)$ which, when given the current assignment of $Var$ as an input, output a value such that:*
   a) *whenever $u$ inserts a node $v$ into $Var$, then $id(v) \in [\gamma^{min}_{u,Var}(A_{Var}), \gamma^{max}_{u,Var}(A_{Var})]$ for the previous assignment $A_{Var}$ of $Var$, and*
   b) *whenever $u$ removes a node $v$ from $Var$, then for the assignment $A_{Var}$ of $Var$ before the execution of that action and for the assignment $A'_{Var}$ of $Var$ after the execution of that action, either $\gamma^{min}_{u,Var}(A'_{Var}) > \gamma^{min}_{u,Var}(A_{Var})$, or $\gamma^{max}_{u,Var}(A'_{Var}) < \gamma^{max}_{u,Var}(A_{Var})$, and*

2. *there is a message type with exactly one parameter holding a node reference (referred to as* DELEGATE*(x)) and whenever a node $u$ removes a search edge $(u, v)$ or does not keep a reference of a node $v$ received in a* DELEGATE*(v) message, $u$ delegates $v$'s reference in a* DELEGATE*(v) message to a search neighbor $w$ such that* $\min(id(u), id(v)) < id(w) < \max(id(u), id(v))$.

Informally speaking, the first sub-property makes sure that $[\gamma^{min}_{u,V}(A), \gamma^{max}_{u,V}(A)]$ monotonically shrinks and implies that each search edge can be removed from the neighborhood only finitely often. The second sub-property intuitively means that a node $v$ is always delegated to a neighbor $w$ that is closer to $v$ than the current node $u$, yet on the same side of $v$ as $u$ (in terms of the total order of the identifiers). Recall that Theorem 2.2 implies that in any protocol that does not disconnect the network graph, each node always needs to delegate a reference it does not keep (i.e., it cannot simply throw away this reference). Furthermore, note that the MCP is generally not a severe restriction. Most existing protocols that stabilize to feasible topologies naturally fulfill this property (see Section 4.5). To simplify the description, in the following we assume that there is exactly one fast search protocol for each topology, which we refer to as *the* fast search protocol for $T$ and thus omit its mention when we use the notion of a feasible stabilization protocol.

Given these definitions we can now formally state the problem we consider in this chapter. The main goal is to develop a generic protocol that satisfies monotonic searchability according to a small set of invariants in feasible topologies for which a self-stabilizing protocol fulfilling the MCP exists. In addition, we show that if a fast search protocol for such a topology has a running time of $T(n)$ then our protocol answers successful search requests in time $O(T(n))$ once the topology has been stabilized.

## 4.2. Primitives for Monotonic Searchability

Although the primitives of [KSS17] are general enough to construct any conceivable overlay, they do not inherently allow for monotonic searchability. This is due to the fact that the delegation primitive may replace an explicit search edge $(u, v)$ by a path $(u, w, v)$ consisting of an explicit search edge $(u, w)$ and an implicit edge $(w, v)$. Thus a search message from $u$ to $v$ issued after the delegation may be processed by $w$ before there is a path from $w$ to $v$ via explicit search edges, causing the search message to fail (even though an earlier message sent while $(u, v)$ was still an explicit edge was delivered successfully). This is illustrated in Figure 4.2. Consequently, we are going to introduce a new set of primitives that enables monotonic searchability. In addition, we show how to transform existing protocols using the primitives of $\mathcal{ID}\mathcal{F}$ and fulfilling the MCP into protocols using the new set of primitives.

One key aspect of the new primitives is that nodes may not immediately delegate search edges that they want to get rid of. To deal with this, in every fixed state $S$ in every execution of a self-stabilizing protocol, each node $u$ can divide its explicit


(a) Initial situation. (b) Situation after $u$ delegated $(u, v)$ to $w$.

Figure 4.2.: Delegations can interfere with monotonic searchability: In the left image, a search message from $u$ to $v$ can be delivered. If $u$ delegates $(u, v)$ to $w$, whose result is shown in the right image, the same search message could not be delivered.

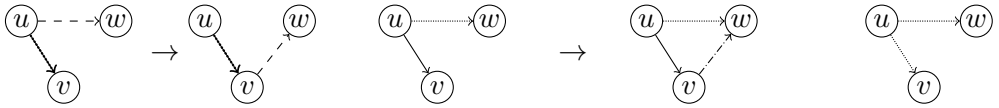
search edges into two subsets: the *stable edges* and the *temporary edges* (not to be confused with implicit edges). The first set contains those explicit search edges that $u$ wants to keep, given its current neighborhood in $S$; the second set holds the explicit search edges that are not needed from the perspective of $u$ in $S$. Note that the set of temporary edges can be empty. We further make use of the following definition:

**Definition 4.7** (Stable / Temporary Neighborhood)**.** *The* stable neighborhood $\Gamma_s(u)$ *of a node $u$ is defined as:* $\Gamma_s(u) := \{v \in \Gamma^*(u) : (u, v)$ *is a stable edge*$\}$*. Likewise, the* temporary neighborhood $\Gamma_t(u)$ *of a node $u$ is defined as:* $\Gamma_t(u) := \{v \in \Gamma^*(u) : (u, v)$ *is a temporary edge*$\}$

For the new primitives, a node does not only store references of its neighbors but additionally stores sequence numbers for every reference in its local memory: i.e., every node $u$ stores for each neighbor $v$ an entry $u.eseq[id(v)]$ (or $u.eseq[v]$, in short).

The new set of primitives is basically obtained by replacing the delegation primitive with a new primitive, called the *safe-delegation primitive*. This works as follows:

**Definition 4.8** (Safe-Delegation Primitive)**.** *Consider a node $u$ that has references of two different nodes $v$ and $w$ and that does not want to keep $w$'s reference. Then* safe-delegation *of the edge $(u, w)$ depends on whether this edge is an implicit, explicit non-search, or explicit search edge: If $(u, w)$ is an implicit or non-search edge, $(u, w)$ is delegated as in the original delegation primitive. If $(u, w)$ is an explicit search edge, the edge is a temporary edge by definition because temporary edges are those explicit search edges a node does not want to keep. To safe-delegate the temporary edge $(u, w)$ to a node $v$, $(u, v)$ must be a stable edge and* $\min(id(u), id(w)) < id(v) < \max(id(u), id(w))$*. $u$ then sends a* DelegateREQ$(u, w, eseq)$ *message to $v$, where $eseq = u.eseq[w]$. Additionally, it sets $u.eseq[v]$ to* $\max\{u.eseq[v], u.eseq[w] + 1\}$*. Any node $v$ that receives a* DelegateREQ$(u, w, eseq)$ *message, adds $(v, w)$ to its set of search edges (either as a stable edge if $v$ wants to keep $w$'s reference or a temporary edge if $v$ does not want to keep $w$'s reference), sets $v.eseq[w]$ to* $\max\{v.eseq[w], eseq + 1\}$ *and sends a* DelegateACK$(w, eseq)$ *message back to $u$. Upon receipt of this message, $u$ checks whether $eseq = u.eseq[w]$ and whether $(u, w)$ is actually a temporary edge (note that the last check is necessary to handle*




(a) An implicit or a non-search edge $(u, w)$ may be delegated via a stable or temporary edge $(u, v)$ as in the original delegation primitive.

(b) To safe-delegate a temporary edge $(u, w)$, $u$ sends $w$'s reference in a DelegateREQ() message to a stable neighbor $v$. $v$ then creates a stable or temporary edge $(v, w)$. Note: Only after the receipt of a corresponding DelegateACK() message may $u$ remove its temporary edge to $w$ and send $w$'s reference to one of its neighbors.

(c) A temporary edge $(u, w)$ cannot be safe-delegated via a temporary edge $(u, v)$.

Figure 4.3.: Rules and restrictions of a safe-delegation.

*corrupted initial states). If both conditions hold, u removes the temporary edge to w and sends w to one of its neighbors in a message. Otherwise, u either keeps w's reference or sends it to one of its neighbors.*

The different cases described in Definition 4.8 are illustrated in Figure 4.3. Furthermore, the safe-delegation primitive is also described in pseudocode in Listing 4.1.

Listing 4.1: Pseudocode of the safe-delegation Primitive

```
1  // Safe-Delegation of an edge (u, w) by a node u to a neighbor v of u
2     if (u, w) is implicit or a non-search edge then
3     |  // w's reference was received in a message
4     |  (u, w) is delegated to v as in the original Delegation primitive
5     if (u, w) is an explicit search edge then
6     |  // when (u, w) is supposed to be delegated, (u, w) is a temporary edge by definition
7     |  send DelegateREQ(u, w, eseq[w]) to v
8     |  eseq[v] := max {eseq[v], eseq[w] + 1}
9
10 // handling of a DelegateREQ(u, w, eseq) message by a node v:
11    either Γ_s := Γ_s ∪ {w} or Γ_t := Γ_t ∪ {w}
12    eseq[w] := max{eseq[w], eseq + 1}
13    send DelegateACK(w, eseq) to u
14
15 // handling of a DelegateACK(w, eseq) message by a node u:
16    if eseq = eseq[w] and w ∈ Γ_t then
17    |  Γ_t := Γ_t \ {w}
18    |  send w to a neighbor in an arbitrary message (excluding DelegateREQ() or
19    |    DelegateACK())
20    else
21    |  either keep w's reference in Γ_s or send w to a neighbor in an arbitrary message (excluding
22    |    DelegateREQ() or DelegateACK())
```

We define $\mathcal{ISF}$ as the set consisting of the three primitives introduction, safe-delegation and fusion. Throughout the chapter we assume that DelegateREQ()

and DelegateACK() messages are sent only in the context of the safe-delegation primitive and that the $u.eseq[v]$ variables for all nodes $u$ and $v$ are changed only in the context of the safe-delegation primitive and only in the way described there. Let $\mathcal{P}_{\mathcal{ISF}}$ denote the set of all distributed protocols in which all interactions between nodes can be decomposed into the primitives of $\mathcal{ISF}$.

### 4.2.1. Universality of the New Primitives

To show that our primitives are search-universal we first show that they are weakly universal.

**Lemma 4.9.** *$\mathcal{ISF}$ is weakly universal.*

*Proof.* The idea of this proof similar to that of the proof of (weak) universality for the original primitives [KSS17]. However, it is a bit more involved because of the safe-delegation primitive. In general, we will present a simple general strategy of how to transform an arbitrary weakly connected graph $G = (V, E)$ into any other strongly connected graph $G' = (V, E')$. At first, we use the introduction primitive to transform $G$ into a clique: i.e., every node continuously introduces all neighbors (including itself) to each other. Obviously, $O(\log n)$ communication rounds are sufficient to build a clique as the distances between the nodes halve in each round of these introductions.

Next we show that by using safe-delegation and fusion one can transform the clique into $G' = (V, E')$. In general, each node $u$ always keeps all references of all other nodes $w$ with $(u, w) \in E'$. Furthermore, whenever a node $u$ has two references of the same node $w$ it fuses them. For the remaining superfluous edges, consider the following approach: For every edge $(u, w)$ that is in the clique but not in $E'$, each node $u$ uses the safe-delegation primitive to delegate the reference of $w$ to the neighbor $v$ that is the next node on a straight path from $u$ to $w$ in $E'$ (in the following, we use the notation $P_{E'}(u, w)$ for this path). Note that such a path exists according to the definitions of a feasible topology (c.f. Definition 4.1) and a fast search protocol (c.f. Definition 4.2). More specifically, $u$ sends a DelegateREQ$(u, w, eseq)$ message to $v$ and when it receives a DelegateACK$(w, eseq)$ message, it no longer stores the reference of $w$, but delegates it to $v$ (which is still a neighbor of $u$ by what we said before) via a Delegate$(w)$ message. We show that by this procedure, all edges $(u, v) \notin E'$ will eventually vanish.

We define the potential $X := \sum_{u \in V} X(u)$. For every node $u \in V$, $X(u) := \max_{w \in V:(w,u) \in NG \setminus E'} |P_{E'}(w, u)|$. Note that no node $x$ ever delegates an edge $(x, z)$ to a node $y$ such that $|P_{E'}(y, z)| > |P_{E'}(x, z)|$. Thus, $X$ never increases. We show that $X$ monotonically decreases. For an arbitrary node $w$, consider a node $u \in \text{argmax}_{u' \in V:(u',w) \in E}(|P_{E'}(u', w)|)$ and let $d := |P_{E'}(u, w)|$. According to the procedure we describe, $u$ sends a DelegateREQ$(u, w, eseq)$ with $eseq = u.eseq[w]$ message to $v$, the next node on $P_{E'}(u, w)$. By the fair message receipt assumption, $v$ will receive this message, act according to the safe-delegation primitive and send

back a DELEGATEACK$(w, eseq)$ message to $u$ which will ultimately be received by $u$. Note that $u.eseq[w] = eseq$ then because the only occasion at which $u$ would have increased $u.eseq[w]$ since the sending of the DELEGATEREQ$(u, w, eseq)$ message is that it received a DELEGATEREQ$(x, w, eseq')$ message for some node $x$ and some number $eseq' \geq eseq$, but this would imply $(x, w) \in NG$ when that message was sent and $|P_{E'}(x, w)| > P_{E'}|(u, w)|$, yielding a contradiction to the definition of $u$. Thus, $u$ will no longer store the reference of $w$ and forward it via a DELEGATE$(w)$ message to $v$. As soon as this has happened for every $u'$ with $|P_{E'}(u', w)| = d$, $X$ decreases. By induction we have that $X$ will eventually be 0, which finishes our proof. □

### 4.2.2. Invariants for Safe-Delegation

The primitives in $\mathcal{P}_{\mathcal{ISF}}$, or, more specifically, the safe-delegation primitive will significantly help to enable monotonic searchability. Intuitively, it will ensure that if a node $v$ is reachable from a node $u$ via a straight path in some state, then the same will hold in every later path as well. However, the safe-delegation primitive introduces two new message types that may, as every type of message, be corrupted initially. Therefore, we now define two **message invariants**:

**Definition 4.10** (Invariants Concerning the Messages Required for Safe-Delegation)**.** *We define the following two invariants, in which we require that $u$, $v$ and $w$ are node references and eseq is an integer greater than or equal to* 0*.*

1. *If there is a* DELEGATEREQ*$(u, w, eseq)$ message in $v.Ch$, then (i) there exists a straight path $P = (u = x_1, x_2, \ldots, x_k = v)$ that does not contain $(u, w)$, such that for every $1 \leq i < k$, $x_i.eseq[x_{i+1}] > u.eseq[w]$, or (ii) $u.eseq[w] > eseq$.*

2. *If there is a* DELEGATEACK*$(w, eseq)$ message in $u.Ch$, then (i) there exists a straight path $P = (u = x_1, x_2, \ldots, x_k = w)$ that does not contain $(u, w)$, such that for every $1 \leq i < k$, $x_i.eseq[x_{i+1}] > u.eseq[w]$, or (ii) $u.eseq[w] > eseq$.*

Intuitively, Invariant 1 states that whenever a node $v$ has a DELEGATEREQ$(u, w,$ $eseq)$ message in $v.Ch$ (i.e., node $u$ asked $v$ to establish the edge $(v, w)$ such that $u$ may remove its own edge to $w$), then there is a straight path from $u$ to $v$ that does not use the edge $(u, w)$. Invariant 2 states that whenever a node $u$ has a DELEGATEACK$(w, seq)$ message in $u.Ch$ (i.e., some other node $v$ which $u$ asked to establish the edge $(v, w)$ has already done so), then there is a straight path from $u$ to $w$ that does not use the edge $(u, w)$. However, both statements need to hold only if the value of ESEQ indicates that the messages belong to a current safe-delegation: i.e., if $u.seq[w] > eseq$, the DELEGATEREQ() or DELEGATEACK() message can be ignored.

In the following, we say a state $S$ is *safe* if Invariant 1 and Invariant 2 of Definition 4.10 hold for all DELEGATEREQ() and DELEGATEACK() messages in $S$. In order to approach search universality, we prove the following lemma.

**Lemma 4.11.** *Consider an arbitrary computation $C$ of a protocol $P \in \mathcal{P}_{\mathcal{ISF}}$ that contains at least one safe state. Let $S$ be the first safe state in $C$. For every state $S' \geq S$ it holds:*

1. *$S'$ is safe, and*
2. *for every two nodes $u$ and $v$ such that $v \in R(u, ID)$ in $S'$ for some identifier $ID$, $v \in R(u, ID)$ in every state $S'' \geq S'$.*

*Proof.* For an arbitrary computation $C$ that contains a safe state, let $S$ be the first safe state. We prove 1 inductively by showing that every state subsequent to a safe state is safe as well. Let $S'$ be an arbitrary safe state.

We begin with showing that Invariant 1 of Definition 4.10 will be true in the state subsequent to $S'$ as well. Note that since a DelegateREQ() message is only sent as specified in the Safe Delegation primitive, we know that if a new DelegateREQ$(u, w, eseq)$ message is sent from node $u$ to node $v$, then there has to be an explicit edge $(u, v)$, $u.eseq[v]$ is set to at least $u.eseq[w] + 1$, and $\min(id(u), id(v)) < id(w) < \max(id(u), id(v))$ (see Definition 4.8). Thus, new DelegateREQ() messages cannot violate Invariant 1. Additionally, for every pair of nodes $u$ and $v$, $u.seq[w]$ is never decreased. Thus, as long as the graph does not change, no existing DelegateREQ$(v, w, eseq)$ message such that $u.seq[w] > eseq$ can cause Invariant 1 to be violated. Therefore, the only other option under which the first invariant could become false is that for an existing DelegateREQ$(u, w, eseq)$ message in $v.Ch$ for a node $v$ such that $u.eseq[w] \leq eseq$ an explicit search edge $(x, y)$ on the only remaining straight path $P$ from $u$ to $v$ that does not contain $(u, w)$ is removed. According to the definition of $\mathcal{P}_{\mathcal{ISF}}$ and, in particular, the definition of the safe-delegation primitive (see Definition 4.8), an explicit search edge is removed only if $x$ receives a DelegateACK$(y, eseq')$ message and $eseq' = x.eseq[y]$. For this message, the second invariant holds by assumption, yielding the existence of a straight path $Q = (x = x_1, x_2, \ldots, x_k = y)$ even after the removal of $(x, y)$ such that for every $1 \leq i < k : x_i.eseq[x_{i+1}] > x.eseq[y]$. Notice that since $P$ was the only remaining straight path from $u$ to $v$ not via $(u, w)$, the first invariant implies that $x.eseq[y] > u.eseq[w]$. Thus, there is a straight path $R = (u = y_1, y_2, \ldots, y_l = w)$, which is $P$ with $(x, y)$ being replaced by $Q$, such that for all $1 \leq i < l : y_i.eseq[y_{i+1}] > u.eseq[w]$. In particular, this implies that $(u, w)$ is not contained in $R$. Thus, the first invariant still holds.

Similarly, to show that Invariant 2 holds in the state subsequent to $S'$, note that a new DelegateACK$(w, eseq)$ message is sent to a node $u$ only as a response to a DelegateREQ$(u, w, eseq)$ message received by a node $v$. According to Invariant 1, if that is the case then there exists a straight path $P = (u = x_1, x_2, \ldots, x_k = v)$ that does not contain $(u, w)$ and $x_i.eseq[x_{i+1}] > u.eseq[w]$ for every $1 \leq i < k$ (or $u.eseq[w] > eseq$, in which case we are done). Since $v$ adds a new explicit edge $(v, w)$ when it sends out the DelegateACK$(w, eseq)$ message and also ensures that $v.eseq[w] > eseq$ (by adjusting $v.eseq[w]$), Invariant 2 still holds afterwards. Since no node $u$ decreases a $u.seq[w]$ value for any node $w$, the only option left

under which Invariant 2 could become false for an existing DELEGATEACK$(w, eseq)$ message in $u.Ch$ such that $u.seq[w] \leq eseq$ for some node $u$ is that an edge $(x, y)$ on the only remaining straight path $P$ from $u$ to $w$ that does not contain $(u, w)$ is removed or becomes implicit. However, for analogous reasons as for Invariant 1 above, there then exists another straight path $R = (u = y_1, y_2, \ldots, y_l = w)$ from $u$ to $w$ not containing $(u, w)$ such that for all $1 \leq i < l : y_i.eseq[y_{i+1}] > u.eseq[w]$. Therefore the second invariant holds as well. Consequently, the state subsequent to $S'$ is safe.

Assume that there is an identifier $ID$ and two nodes $u$ and $v$, such that $v \in R(u, ID)$ in some state $S' \geq S$ but $v \notin R(u, ID)$ in some later state. Note that $v \in R(u, ID)$ in $S'$ implies $v \leq ID$ if $id(u) < ID$ and that $v \geq ID$ if $ID < id(u)$. Let $S''$ be the first state such that $v \notin R(u, ID)$ and let $S''_{prev}$ be the state predecessing $S''$. According to Definition 4.5, in $S''_{prev}$, there was a straight path $Q$ from $u$ to $v$. Thus, in the action executed to obtain $S''$, some explicit search edge $(x, y)$ on $Q$ must have been removed. Since $P \in \mathcal{P}_{\mathcal{ISF}}$ this may have happened only if $x$ received a DELEGATEACK$(y, eseq)$ message and $eseq = x.eseq[y]$. Invariant 2 yields that there is still a straight path $P = (x = x_1, x_2, \ldots, x_k = y)$ afterwards. Therefore $Q$ with $(x, y)$ replaced by $P$ is a straight path: i.e., there still exists a straight path from $u$ to $v$ contradicting the fact that $v \notin R(u, ID)$ in $S''$. □

## 4.3. Transforming Classical Protocols

Let $\mathcal{P}_{\mathcal{IDF}}$ denote the set of all distributed protocols in which all interactions between nodes can be decomposed into the primitives of $\mathcal{IDF}$. In this section we show that every protocol $A \in \mathcal{P}_{\mathcal{IDF}}$ that fulfills the MCP and that self-stabilizes to some feasible topology $T$ can be transformed into a protocol $B \in \mathcal{P}_{\mathcal{ISF}}$ for which it holds that in every computation of $B$ there is a computation suffix throughout which Invariants 1-2 hold. More formally:

**Theorem 4.12.** *Consider a protocol $A \in \mathcal{P}_{\mathcal{IDF}}$ that self-stabilizes to a strongly connected feasible topology $T$ and that fulfills the* MCP. *Then $A$ can be transformed into another protocol $B \in \mathcal{P}_{\mathcal{ISF}}$ such that $B$ self-stabilizes to $T$ as well and in every computation of $B$ there exists a computation suffix in which every state is safe.*

*Proof.* The general idea of the transformation of protocol $A$ into protocol $B$ is to replace every use of the Delegation primitive by the original protocol $A$ with a use of the safe-delegation primitive. Therefore, every edge that would be delegated away by $A$ becomes a temporary edge and the reference of its endpoint is sent to the receiving node. If that node wants to keep the received reference, the new edge becomes a stable edge. Otherwise, $A$ would delegate away that reference again and this Delegation would again be transformed into a safe-delegation. As we will see, the MCP ensures that this process will not repeat infinitely often: i.e., at some point in time, the delegated reference will be kept, which then forms a stable edge.

We will show that finally, every computation of $B$ converges to a fixed topology, which we then prove to be equal to the one $A$ converges to. Last, we will prove the closure property of $B$ to finish this proof.

Consider the following transformation of $A$:

1. Every action $a$ in $A$ is transformed into an action $b$ in the following way: Whenever in an action $a$ the current node $u$ removes the last reference of a search neighbor $w$ from its search variables and sends it to a stable neighbor $v$ (which it does in a DELEGATE($w$) message according to sub-property 2 of the MCP), action $b$ makes the edge $(u, w)$ temporary (i.e., it stores $w$'s reference in a dedicated set variable $N_{tmp}$) and additionally sends out a DELEGATEREQ($u, w, u.eseq[w]$) message to $v$. Other than that, $b$ behaves exactly as $a$. In particular, $b$ does not consider the references in $N_{tmp}$ at all (and all computations are performed on stable, non-search, or implicit edges only).

2. A node $v$ that receives a DELEGATEREQ($u, w, eseq$) message executes DELEGATE($w$) and acts according to the safe-delegation primitive. This means it makes $(v, w)$ an explicit search edge (either a stable edge if DELEGATE($w$) would keep $w$'s reference or a temporary edge if that action would immediately delegate the reference again), sets $v.eseq[w]$ to $\max\{v.eseq[w], eseq+1\}$ and sends a DELEGATEACK($w, eseq$) message back to $u$. If $(v, w)$ is a temporary edge after this, it additionally sends a DELEGATEREQ($v, w, v.eseq[w]$) message to the neighbor that DELEGATE($w$) would forward this reference to.

3. A node $u$ that receives a DELEGATEACK($w, eseq$) message first checks whether $eseq = u.eseq[w]$ and whether $(u, w)$ is a temporary edge (i.e., it checks whether an execution of DELEGATE($w$) would not keep $(u, w)$ in $u$'s current state) and, if so, removes the temporary edge $(u, w)$. It then executes DELEGATE($w$).

4. In the TIMEOUT action, in addition to the steps performed in the original TIMEOUT action, every node $u$ for every reference $w$ stored in $u.N_{tmp}$ checks whether $(u, w)$ is a temporary edge: i.e., it checks whether an execution of DELEGATE($w$) would not keep $(u, w)$ in $u$'s current state). If so, $u$ sends a DELEGATEREQ($u, w, u.eseq[w]$) message to the stable neighbor $v$ to which that edge would be delegated to in DELEGATE($w$). Otherwise, $u$ stores the reference in the search variable(s) that DELEGATE($w$) would store them in (making the edge $(u, w)$ a stable edge).

We denote the resulting protocol by $B$. By the construction of $B$ and the definition of $\mathcal{P}_{\mathcal{ISF}}$, $B \in \mathcal{P}_{\mathcal{ISF}}$.

Note that the construction of $B$ and the fact that $A$ fulfills the MCP yields that the following set of properties (not coincidentally resembling the MCP) is satisfied by $B$:

1. for every node $u$ and every search variable $Var$ other than $N_{tmp}$ there are two functions $\gamma^{min}_{u,Var}(A_{Var})$ and $\gamma^{max}_{u,Var}(A_{Var})$ which, when given the current assignment of $Var$ as an input, output a value such that:
   a) whenever $u$ inserts a node $v$ into $Var$, then $id(v) \in [\gamma^{min}_{u,Var}(A_{Var}), \gamma^{max}_{u,Var}(A_{Var})]$ for the previous assignment $A_{Var}$ of $Var$, and
   b) whenever $u$ removes a node $v$ from $Var$, then for the assignment $A_{Var}$ of $Var$ before the execution of that action and for the assignment $A'_{Var}$ of $Var$ after the execution of that action, either $\gamma^{min}_{u,Var}(A'_{Var}) > \gamma^{min}_{u,Var}(A_{Var})$, or $\gamma^{max}_{u,Var}(A'_{Var}) < \gamma^{max}_{u,Var}(A_{Var})$, and
2. whenever a node $u$ removes an edge $(u, v)$ from its set of stable edges or makes $(u, v)$ a temporary edge upon receipt of a DelegateREQ$(u, v, eseq)$ message, $u$ safe-delegates $v$'s reference in a DelegateREQ$(u, v, u.eseq[v])$ message to a stable neighbor $w$ such that $\min(id(u), id(v)) < id(w) < \max(id(u), id(v))$.

Property 1 basically implies that for each node $u$ and every search variable $Var$ other than $N_{tmp}$, $|[\gamma^{min}_{u,Var}(A_{Var}), \gamma^{max}_{u,Var}(A_{Var})]|$ monotonically decreases and eventually the set of stable neighbors of each node will be fixed. Property 2 implies that every temporary edge $(u, v)$ is always delegated to a node between $u$ and $v$.

We now prove the convergence property of $B$. Consider an arbitrary but fixed computation $C_B$ of $B$ starting from an arbitrary weakly connected graph. Let $S$ be the first state of $C_B$ such that the stable neighborhood of each node remains unchanged in all states after $S$. Such a state has to exist due to the above Property 1. We first show that there will be a state $S'$ from which on there will be no temporary edges in the system.

The key idea is that, informally speaking, starting from $S'$ any new temporary edge can only appear as a result of a (Safe)-Delegation of an existing temporary edge. Since every edge is delegated to a node closer to the endpoint according to the above Property 2, this may occur only finitely often. In addition, this causes no new temporary edges to appear as soon as all temporary edges have vanished. To prove this formally, let $g(w) := \max\{\max_{u:(u,w)\ is\ temporary}(|id(u) - id(w)| \cdot 2), \max_{u:\text{DelegateREQ}(v,w,eseq)\in u.Ch}(|id(u) - id(w)| \cdot 2 + 1)\}$ where the two maximum functions are supposed to be 0 if there is no such node $u$ at all. We now define the potential $\Phi := \sum_{w\in V} g(w)$ and show that as long as $\Phi$ is not zero, it will decrease in finite time.

Consider an arbitrary node $w$ such that $g(w) > 0$. First assume $g(w)$ is odd. This implies that there is at least one node $u$ with a DelegateREQ$(v, w, eseq)$ message in $u.Ch$ and $|id(u) - id(w)| \cdot 2 + 1 = g(w)$ and that there is no node $x$ with a temporary edge $(x, w)$ such that $|id(x) - id(w)| > |id(u) - id(w)|$ (by the definition of $g(w)$). Note that this situation not only occurs from corrupted initial states but also if during Timeout of a node several DelegateREQ() messages were sent and the original temporary edge has already been removed in the meantime. Since new temporary edges only result from stable edges turning into temporary edges (which does not happen after $S$ as we argued before) or from receipts of DelegateREQ()

messages, once all the aforementioned DelegateREQ$(v, w, eseq)$ messages have been received by $u.Ch$ and all other nodes with the same distance to $w$, $\Phi$ will decrease. Next assume $g(w)$ is even. This implies that there is at least one node $u$ with distance $g(w)$ and a temporary edge $(u, w)$ and that there is no Delegate$(v, w, eseq)$ message in $u.Ch$ or in the channel of any other node with an equal or higher distance to $w$ nor a node $x$ such that $(x, w)$ is a temporary edge and $|id(x)-id(w)| > |id(u)-id(w)|$. According to the above Property 2 of $B$, $u$ will send a DelegateREQ$(u, w, eseq)$ to a node $v$ such that $|id(v)-id(w)| < |id(u)-id(w)|$. This will be answered with a DelegateACK$(w, eseq)$ message, causing $u$ to remove the explicit edge $(u, w)$ and to forward $w$ to a node closer to $w$ than $u$. The same happens for all other temporary edges $(y, w)$ with $|id(y) - id(w)| \cdot 2 = g(w)$. After the last such edge has been removed, $\Phi$ decreases.

Note that the aforementioned also implies that $\Phi$ never increases. Therefore, $\Phi$ will eventually be and remain zero from some state $S'$, meaning that there will be no more temporary edges from $S'$ on. Since no DelegateREQ() and DelegateACK() message will be sent during or after $S'$, Invariants 1 and 2 hold in this state.

Since there are no DelegateREQ() and DelegateACK() messages in $S'$ and the sets $u.N_{tmp}$ are empty for every node $u$ in $S'$, we can consider $S'$ as an input to algorithm $A$ and let $SUFFIX_A(S')$ denote the computation of $A$ starting from $S'$. Likewise, let $SUFFIX_B(S')$ be the suffix of $C_B$ starting from $S'$ in $B$. Consider the first execution of an action whose result is different in $A$ and $B$. Note that since there are no DelegateREQ() and DelegateACK() messages in $S'$ or any later state and since $u.N_{tmp}$ is empty for every node $u$ throughout $SUFFIX_B(S')$, this action cannot be a DelegateREQ(), DelegateACK() or Timeout message. By the above transformation, the only possibility left is that this action removes the last reference of another node $w$ from a search variable of a node $u$ in $A$ and moves it to $u.N_{tmp}$ in $B$. This, however, cannot be the case as argued before. Therefore, $SUFFIX_A(S')$ and $SUFFIX_B(S')$ are equivalent in $A$ and $B$: i.e., for every $i \geq 1$ the $i$-th state $s_i(A)$ of $SUFFIX_A(S')$ and the $i$-th state $s_i(B)$ of $SUFFIX_B(S')$ are identical except for that each node in $s_i(B)$ has an additional empty set $N_{tmp}$. This implies that $A$ performs the same changes to the network graph of $S'$ as $B$ does. Since $A$ stabilizes to $T$ and $C_B$ was chosen arbitrarily, we obtain that $B$ converges to $T$.

For the closure property, again compare the execution of $A$ and of $B$ starting from a legitimate state and note that for analogous reasons, $B$ also does not perform any changes to the network graph. □

## 4.4. The Generic Search Protocol

In this section we describe a generic search protocol such that every protocol in $\mathcal{P}_{\mathcal{ISF}}$ fulfilling the MCP satisfies monotonic searchability according to that search protocol. We assume that when a node $u$ wants to search for a node with identifier

$ID$, it performs an InitiateNewSearch($ID$) action in which a Search($u, ID$) message is created. The search request is regarded as answered as soon as the Search($u, ID$) message is either dropped, i.e., it *fails*, or is received by the node $w$ with $id(w) = ID$, i.e., it *succeeds*.

We begin with describing the protocol in Section 4.4.1. After that, we introduce some additional definitions and state the main theorems regarding the generic search protocol in Section 4.4.2. These definitions include the definition of admissible states. We prove the existence of these in all computations of protocols that satisfy a certain set of properties in Section 4.4.3. Last, in Section 4.4.4, we prove that starting from admissible states, monotonic searchability is satisfied.

### 4.4.1. Protocol Description

We begin with a textual description of the generic search protocol. For a better overview, in Listing 4.2 we specify the pseudocode of this protocol along with supplementary details. The principle idea of the *generic search protocol* is the following: A node $u$ with a Search($u, destID$) message does not directly forward this message through the network. Instead, $u$ buffers the message and initiates a probing algorithm whose goal is to either receive the reference of the node $w$ with $id(w) = destID$ or to get a negative response in case this node does not exist or cannot be reached yet. In the former case, $u$ directly sends Search($u, destID$) to $w$. In the latter case, $u$ drops Search($u, destID$). Whenever an additional Search($u, destID$) message for the same identifier $destID$ is initiated at $u$ while a probing for $destID$ is still in progress, this message is combined with previous Search($u, destID$) messages waiting at $u$. In addition to the aforementioned, an attempt is made to fast search for $destID$ via the fast search protocol for the topology using only stable edges. If a node with identifier $destID$ exists, it might be found this way. However, it is not guaranteed to be found as long as the topology is not stabilized. Therefore, in case the fast search fails, the probing is still continued. This can be used to speed up successful searches in the stabilized topology.

For the probing, a node $u$ with a buffered Search($u, destID$) message periodically initiates a new Probe() message in its Timeout action. This Probe() message contains four arguments: First, a reference *source* of the source of the Probe() message: i.e., a reference of $u$. Second, the identifier $destID$ of the node that is searched. Third, a set $Next$ that holds references of all stable or temporary neighbors of $u$ whose identifier is between $id(u)$ and $destID$ (according to the total order on the identifiers). Last, a sequence number $seq$ that is used to distinguish probe messages that belong to different probing processes from the same node and for the same target, i.e., $seq = u.seq[destID]$, where $u.seq[destID]$ is a value stored at $u$. This is necessary because in each execution of the Timeout action a new probe message is sent, although upon receipt of the first response to such a message, the set of buffered search messages is sent out to the target or dropped completely. Thus, future replies may arrive afterwards and $u$ has to

know that these are outdated. To make this possible, as long as a fixed search message $m$ for $destID$ is buffered at $u$, $u.seq[destID]$ will not change its value. However, once $m$ has been dropped or delivered, as soon as the next search message for $destID$ is initiated, $u.seq[destID]$ will increase. All in all, $u$ initiates a PROBE$(source, destID, Next, seq)$ message and sends this message to the node in $Next$ closest to $u$ (observe that this is the node whose identifier has the maximum distance to $destID$).

Any intermediate node $v$ that receives a PROBE$(source, destID, Next, seq)$ message first checks whether $id(v) = destID$. If so, $v$ sends a reference of itself to $source$ via a PROBESUCCESS$(destID, dest)$ message with $dest = v$. Otherwise, $v$ removes itself from $Next$ and adds all its (stable or temporary) neighbors to $Next$ whose identifier is between $id(u)$ and $destID$ (according to the total order on the identifiers). If $Next$ is empty after this step, $v$ responds to $source$ via a PROBEFAIL$(destID, seq)$ message. Otherwise, $v$ forwards the PROBE$(source, destID, Next, seq)$ message (with the already described changes performed to $Next$) to the node in $Next$ whose identifier has the maximum distance to $destID$. If the initiator $u$ of a probe receives a PROBESUCCESS$(destID, dest)$ message, $u$ sends out all (possibly combined) SEARCH$(u, destID)$ messages waiting at $u$ to $dest$ (thus stopping the probing). If, however, $u$ receives a PROBEFAIL$(destID, seq)$ message, $u$ first checks whether $seq = u.seq[destID]$: i.e., it checks whether the received message is a response to the current batch of search requests. Without this check, a PROBEFAIL$(destID, seq)$ message belonging to an earlier batch of buffered messages could cause a currently ongoing probing to be aborted. If $seq = u.seq[destID]$ is true, $u$ drops all SEARCH$(u, destID)$ messages waiting at $u$ (thus also stopping the probing). Otherwise, $u$ simply drops the received PROBEFAIL$(destID, seq)$ message.

Listing 4.2: The Generic SEARCH protocol

```
1  INITIATENEWSEARCH(destID)
2     create a new message m = SEARCH(self, destID)
3     if WaitingFor[destID] = ∅ then
4     |  seq[destID] := seq[destID] + 1 // seq[destID] is assumed to be -1 if its
5     |                                 // current value is not a positive integer
6     |  if there is a next hop w for the search for destID using the fast
7     |    search protocol on the stable edges then
8     |  |  send FASTPROBE(self, destID) to w
9     // store the message in WaitingFor
10    WaitingFor[destID] := WaitingFor[destID] ∪ {m}
11
12 TIMEOUT // suffix to the existing timeout action of the protocol
13    for every destID such that WaitingFor[destID] ≠ ∅ do
14    |  Next := {w ∈ Γ_s ∪ Γ_t : min(id(self), destID) < id(w) < max(id(self), destID)}
15    |  send PROBE(self, destID, Next, seq[destID]) to the closest node to self in Next
16
17 PROBE(source, destID, Next, seq)
18    if destID = id(self) then
19    |  if Next ≠ ∅ then // can only occur in initial states
20    |  |  for all u ∈ Next
```

```
21      | | | send DELEGATE(u) to self
22      | send PROBESUCCESS(destID, self) to source
23      else // destID ≠ id(self)
24      | Next := Next \ {self}
25      |          ∪{w ∈ Γ_s ∪ Γ_t : min(id(self), destID) < id(w) < max(id(self), destID)}
26      | if Next = ∅ then
27      | | send PROBEFAIL(destID, seq) to source
28      | | send DELEGATE(source) to self // required to maintain connectivity
29      | else // Next ≠ ∅
30      | | u := argmax_{v∈Next}{id(v) − destID}
31      | | if u is not a neighbor of self then
32      | | | send IMPLDELEGATE(u) to self // required to maintain connectivity
33      | | send FORWARDPROBE(source, destID, Next, seq) to u
34
35  PROBESUCCESS(destID, dest)
36      send all m ∈ WaitingFor[destID] to dest
37      WaitingFor[destID] := ∅
38      send DELEGATE(dest) to self
39
40  PROBEFAIL(destID, seq)
41      if seq = seq[destID] then
42      | // the message belongs to the current set of buffered search requests for destID
43      | WaitingFor[destID] := ∅
44
45  FASTPROBE(source, destID)
46      if destID = id(self) then
47      | send PROBESUCCESS(destID, self) to source
48      else
49      | if there is a next hop w for the search for destID using the fast search protocol on the
50      |  stable edges then
51      | | send FASTPROBE(self, destID) to w
52      | // otherwise, the fast search is just stopped
```

Note that the pseudocode in Listing 4.2 contains some additional statements, which are required for the protocol to operate in accordance to the primitives in $\mathcal{P}_{\mathcal{ISF}}$ and thereby maintain connectivity in spite of corrupted initial states. They assume the existence of a message type DELEGATE($w$) taking exactly one (node reference) parameter that is used to delegate implicit edges until they are either fused with existing edges or used to establish edges that belong to the topology (this can be achieved easily, e.g., by sending them via the path that the fast search protocol for $id(w)$ would take, which is why we do not pay any further attention to this here).

### 4.4.2. Definitions and Main Results

As the generic search protocol cannot guarantee to function properly under the presence of corrupted messages, we define additional invariants that are maintained throughout the executions of the generic search protocol (that did not start with corrupted messages). Note that we will use them in conjunction with the invariants from Definition 4.10, which is why we start with the enumeration at 3.

**Definition 4.13** (Additional Invariants Concerning the Generic Search Protocol's Messages)**.** *We define the following invariants in which we require that source and dest are node references, destID is an identifier, seq is an integer greater than or equal to* 0*, and Next is a (possibly empty) set of node references.*

3. *If there is a* Probe*(source, destID, Next, seq) message in u.Ch, then*
   a) $u \in Next$ *and for every* $w \in Next$, $\min(id(u), destID) \leq id(w) \leq \max(id(u), destID)$,
   b) $R(Next, destID) \subseteq R(source, destID)$ *and*
   c) *if there is a node v such that* $id(v) = destID$ *and* $v \notin R(Next, destID)$*, then for every admissible state with* $source.seq[destID] < seq$, $v \notin R(source, destID)$.
4. *If there is a* FastProbe*(src, destID) message in u.Ch, then* $u \in R(src, destID)$.
5. *If there is a* ProbeSuccess*(destID, dest) message in u.Ch, then* $id(dest) = destID$ *and* $dest \in R(u, destID)$.
6. *If there is a* ProbeFail*(destID, seq) message in u.Ch, then if v exists such that* $id(v) = destID$*, for every admissible state with* $u.seq[destID] < seq$, $v \notin R(u, destID)$.
7. *If there is a* Search*(v, destID) message in u.Ch, then* $id(u) = destID$ *and* $u \in R(v, destID)$.

With these and the aforementioned invariants, we are finally able to give the definition of an admissible state for our problem:

**Definition 4.14** (Admissible State)**.** *A state S is called* admissible *if and only if all of Invariants 1-2 (c.f. Definition 4.10) as well as Invariants 3-7 (c.f. Definition 4.13) hold in S.*

We will also use the notion of a *monotonic-searchability-sufficient* (*ms-sufficient*) protocol:

**Definition 4.15** (Monotonic-Searchability-Sufficient (*ms*-sufficient))**.** *A protocol for the self-stabilization of a topology is* monotonic-searchability-sufficient (*ms*-sufficient) *if and only if*

1. *all interactions between nodes can be decomposed into the primitives in* $\mathcal{ISF}$*,*
2. *it uses the generic search protocol for searching,*
3. *no* Probe*(),* ProbeSuccess*(),* ProbeFail*(),* Search*() or* FastProbe*() message is sent at any other occasion than the ones specified in the generic search protocol, and*

*4. in every computation of the protocol there is a safe state.*

Theorem 4.12 implies the following:

**Corollary 4.16.** *Any conventional protocol $A \in \mathcal{P}_{\mathcal{IDF}}$ that self-stabilizes to a strongly connected topology $T$ and that fulfills the* MCP *can be transformed into an ms-sufficient protocol that self-stabilizes to $T$ as well.*

In the rest of this section, we prove the following theorem:

**Theorem 4.17.** *Every ms-sufficient protocol satisfies monotonic searchability according to Invariants 1-7.*

On a side note, the following result follows directly from the description of the generic search protocol:

**Corollary 4.18.** *Consider an ms-sufficient protocol $P$ that self-stabilizes to a certain topology $T$ and whose fast search protocol has a running time of at most $T(n)$. Then, in legitimate states, $P$ answers successful search requests in time $O(T(n))$.*

### 4.4.3. Proving That Every Computation Has an Admissible Suffix

As a starting point of the proof of Theorem 4.17, we first prove the following lemma:

**Lemma 4.19.** *In every computation $C$ of an ms-sufficient protocol, for every admissible state $S$ in $C$, every state $S' \geq S$ is admissible. Furthermore, every computation of an ms-sufficient protocol contains an admissible state.*

The following sequence of lemmas constitutes the proof of Lemma 4.19. We say a message $m$ *causes* a message $m'$ if the message $m'$ was sent in an action handling the receipt of $m$ or a message caused by $m$.

**Lemma 4.20.** *In every computation of an ms-sufficient protocol, if the first four invariants hold in a state $S$, they hold in every state $S' \geq S$. Furthermore, in every computation of an ms-sufficient protocol, there exists a state in which the first four invariants hold.*

*Proof.* Consider a state $S''$ in which the first two invariants hold. We show that the following six statements hold in $S''$ and every subsequent state:

1. Every PROBE() message sent in the TIMEOUT action conforms to Invariant 3.
2. Every FASTPROBE() message sent in the INITIATENEWSEARCH() action conforms to Invariant 4.
3. Every PROBE() message sent in the action executed on receipt of a PROBE() message conforms to Invariant 3a).

4. Every PROBE() message sent in the action executed on receipt of a PROBE() message that conforms to Invariant 3b) and Invariant 3c) also conforms to Invariant 3b) and Invariant 3c).

5. Every FASTPROBE() message sent in the action executed on receipt of a FASTPROBE() message that conforms to Invariant 4 also conforms to Invariant 4.

6. Every PROBE() message violating the third invariant can cause at most a finite number of PROBE() message that violate Invariant 3b) or 3c).

For the first statement, observe that by construction the protocol ensures that every PROBE($source, destID, Next, seq$) sent in TIMEOUT conforms to Invariant 3a) and 3b) by construction and by Definition 4.5. Note that for every such message $seq = source.seq[destID]$ holds. It is easy to check that $source.seq[destID]$ is monotonically increasing for every node $u$ and every $destID$. Since by construction $R(Next, destID) = R(source, destID)$ and because of Lemma 4.11, for every node $v$, $v \notin R(Next, destID)$ implies that $v \notin R(source, destID)$ in every previous admissible state and, in particular, every admissible state with $source.seq[destID] < seq$. For similar reasons and the definition of the fast search protocol (see Definition 4.2), the second statement also holds.

The third statement follows directly from the protocol.

For the fourth statement, consider an arbitrary PROBE($source, destID, Next, seq$) message received by a node $w$ that conforms to Invariant 3b) and Invariant 3c) and that causes the sending of a new PROBE($source, destID, Next', seq$) message (for some set $Next'$). According to the protocol, $R(Next', destID) = R(Next, destID) \setminus \{w\}$, thus $R(Next', destID) \subseteq R(Next, destID) \subseteq R(source, destID)$ (the last set relationship is due to Invariant 3b)) and Invariant 3b) holds for the new message. Now assume that $v$ exists such that $id(v) = destID$ and $v \notin R(Next', destID)$. If $w = v$, $w$ would not have sent the new message, thus we may assume $w \neq v$. This yields $v \notin R(Next, destID)$, since $R(Next', destID) = R(Next, destID) \cup \{w\}$. By Invariant 3c) of the message received at $w$, we have that Invariant 3c) also holds for the message sent from $w$.

For the fifth statement, consider an arbitrary FASTPROBE($source, destID$) message received by a node $w$ that conforms to Invariant 4 and that causes the sending of a new FASTPROBE($source, destID$) message to some node $v$. Since $v$ is then a neighbor of $w$ whose identifier is between $id(w)$ and $destID$, inclusive, it holds that $v \in R(w, destID)$. By Invariant 4, $w \in R(source, destID)$, thus by definition also $v \in R(source, destID)$. By Lemma 4.11, this can never be rendered untrue in any subsequent state.

For the sixth statement, note that every PROBE($source, destID, Next, seq$) message received at a node $v$ causes at most one new PROBE($source, destID, Next', seq$) message sent by $v$ with the property that $\max_{u' \in Next'} |id(u') - destID| < \max_{u \in Next} |id(u) - destID|$ by construction. Since the number of nodes is finite,

this implies the PROBE($source, destID, Next, seq$) message can cause only a finite number of PROBE() message, which finishes the proof of the sixth statement.

For the first claim of the lemma, assume there is a state $S$ such that the first four invariants hold. Then the first five statements and Lemma 4.11 yield that they will hold in every state $S' \geq S$.

For the second claim of the lemma, note that a state $S''$ in which the first two invariants hold exists in every computation of an *ms*-sufficient protocol (according to the definition of an *ms*-sufficient protocol). The six statements yield that PROBE() or FASTPROBE() messages initiated in TIMEOUT and messages conforming to Invariant 3 or 4 can only cause new messages conforming to Invariant 3 or 4, respectively, and that all other messages will eventually be gone. Thus, there is a state $S''' \geq S''$ such that all messages conform to Invariant 3 and 4. □

**Lemma 4.21.** *In every computation of an ms-sufficient protocol, if the first six invariants hold in a state $S$, they hold in every state $S' \geq S$. Furthermore, in every computation of an ms-sufficient protocol, there exists a state in which the first six invariants hold.*

*Proof.* Consider a state $S''$ in which the first four invariants hold. By Lemma 4.20, they will also hold in every subsequent state. We show that in $S''$ and every subsequent state, no receipt of a PROBE() message causes the sending of a PROBESUCCESS() or PROBEFAIL() message that violates Invariant 5 or Invariant 6, respectively. Observe that according to the protocol, no other action causes a node to send a PROBESUCCESS() or PROBEFAIL() message.

First we show that no receipt of a PROBE() or a FASTPROBE() message can cause the sending of a PROBESUCCESS() message that violates Invariant 5. To be more specific, consider a node $v$ that receives a PROBE($source, destID, Next, seq$) message and this causes $v$ to send a PROBESUCCESS($destID', dest'$) message to a node $u$. In this case, according to the protocol, $id(v) = destID$, $destID' = destID$, $dest' = v$, and $u = source$ must hold. By Invariant 3a), when $v$ receives the PROBE() message, $v \in Next$ and by Invariant 3b), $R(Next, destID) \subseteq R(source, destID)$, i.e., $v \in R(source, destID)$. Thus, the PROBESUCCESS() message sent by $v$ (to *source*) conforms to Invariant 5. For the other case, assume that a node $v$ receives a FASTPROBE($source, destID$) message and that this causes $v$ to send a PROBESUCCESS($destID', dest'$) to a node $u$. In this case, according to the protocol also $id(v) = destID$, $destID' = destID$, $dest' = v$, and $u = source$ must hold. By Invariant 4, when $v$ receives the FASTPROBE() message, $v \in R(source, destID)$. Thus, the PROBESUCCESS() message sent by $v$ (to *source*) conforms to Invariant 5 in this case, too.

Second we show that no receipt of a PROBE() message can cause the sending of a PROBEFAIL() message that violates Invariant 6. Again, consider a node $v$ that receives a PROBE($source, destID, Next, seq$) message causing $v$ to send a PROBEFAIL($destID', seq'$) message to a node $u$. According to the protocol $destID' = destID$, $seq' = seq$ and $u = source$ must hold. Additionally, since

Invariant 3a) holds and because the protocol sends a ProbeFail() message, $R(Next, destID) \setminus \{v\} = \emptyset$. If there is a node $w$ such that $id(w) = destID$, then since Invariant 3c) holds for the Probe$(source, destID, Next, seq)$ message received by $v$, for every admissible state $u.seq[destID] < seq$, $w \notin R(u, destID)$. Thus, the ProbeFail() message sent by $v$ conforms to Invariant 6.

For the first claim of the lemma, assume there is a state $S$ such that the first six invariants hold. By what we showed above, they will also hold in every state $S' \geq S$.

For the second claim of the lemma, note that a state $S''$ as defined above exists in every computation of an *ms*-sufficient protocol by Lemma 4.20. Further note that we showed that no new message can violate Invariant 5 or Invariant 6. Thus, consider the state $S''' \geq S''$ in which all ProbeSuccess() or ProbeFail() messages that violate Invariant 5 or Invariant 6 and that are in channels in state $S''$ have been received. $S'''$ is a state in which the first six invariants hold. □

Finally, we can use these results to prove Lemma 4.19, which we restate as follows:

**Lemma 4.19.** *In every computation $C$ of an ms-sufficient protocol, for every admissible state $S$ in $C$, every state $S' \geq S$ is admissible. Furthermore, every computation of an ms-sufficient protocol contains an admissible state.*

*Proof.* Consider a state $S''$ in which the first six invariants hold. By Lemma 4.21, they will also hold in every subsequent state. We show that in $S''$ and every subsequent state, no Search() message that violates Invariant 7 can be sent.

According to the protocol, the only occasion at which a Search$(v, destID)$ message is sent to a node $u$ is if node $v$ received a ProbeSuccess$(destID', dest)$ message with $destID' = destID$ and $u = dest$. Invariant 5 yields that $id(dest) = destID'$, implying $id(u) = destID$, and $dest \in R(v, destID')$, implying $u \in R(v, destID)$. This implies that every new message conforms to Invariant 7, which completes the proof of the first claim of the lemma.

For the second claim of the lemma, note that a state $S''$ as defined above exists in every computation of an *ms*-sufficient protocol by Lemma 4.21. Consider the state $S \geq S''$ in which all Search() messages violating Invariant 7 that are in channels in $S''$ have been received. $S$ is a state such that all seven invariants hold: i.e., $S$ is an admissible state. □

### 4.4.4. Proving the Correctness in Admissible States

The following sequence of lemmas gives results that hold in admissible states. They turn out to be crucial to prove Theorem 4.17. We begin with proving the following lemma:

**Lemma 4.22.** *In every computation of an ms-sufficient protocol, if a node $u$ has a* Search*$(u, destID)$ message buffered at $u$ in an admissible state, then this message will eventually be delivered or dropped. In the former case, it will be delivered to*

*the node $w$ with $id(w) = destID$ (which exists in this case). In the latter case, either there is no node with identifier $destID$ or all previous* SEARCH*$(u, destID)$ messages that were initiated before that message and that were buffered at $u$ during at least one admissible state have been or will be dropped as well.*

Recall that a SEARCH$(u, destID)$ is *buffered at $u$* if this message is contained in $u.WaitingFor[destID]$.

Before we prove Lemma 4.22, we prove another lemma that will turn out very helpful for that proof:

**Lemma 4.23.** *In every computation of an ms-sufficient protocol, if a node $v$ initiates a* PROBE*$(v, destID, Next, seq)$ message in an admissible state, then $v$ eventually receives either a* PROBESUCCESS*$(destID, dest)$ message for some node $dest$ or a* PROBEFAIL*$(destID, seq)$ message.*

*Proof.* For an arbitrary computation of an $ms$-sufficient protocol, let $S$ be an arbitrary admissible state. In the following, we use the potential function $\Psi(U, ID)$ defined as: $\Psi(U, ID) := \sum_{u \in U} n^{|id(u) - ID|}$, in which $n$ is the total number of nodes.

First of all, note that in $SUFFIX(S)$ if a node $u$ receives a PROBE$(v, destID, Next, seq)$ message such that $R(Next, destID) \setminus \{u\} \neq \emptyset$ then $u$ sends a PROBE$(v, destID, Next', seq)$ message to some other node such that $\Psi(Next', destID) < \Psi(Next, destID)$. This is due to the protocol and because when $u$ receives a PROBE$(v, destID, Next, seq)$ message, $u$ chooses $Next'$ as $Next \setminus \{u\}$ augmented by neighbors whose identifier is between $id(u)$ and $distID$. Thus, $\Psi(Next', destID) < \Psi(Next, destID)$.

Second, the aforementioned and the construction of the protocol give that each PROBE$(v, destID, Next', seq)$ message that does not cause a PROBESUCCESS() or PROBEFAIL() message causes a PROBE$(v, destID, Next'', seq)$ message with $\Psi(Next'', destID) < \Psi(Next', destID)$. Besides, by definition, $\Psi(Next', destID)$ cannot increase for an existing PROBE$(v, destID, Next', seq)$ message. By induction and because $\Psi$ is bounded from below, this yields that there must a node $w$ that does not send a new PROBE() message upon receipt of a PROBE$(v, destID, Next', seq)$ message caused by the original PROBE$(v, destID, Next, seq)$ message sent by $v$. Instead, according to the protocol, it sends either a PROBESUCCESS$(destID, w)$ message or a PROBEFAIL$(destID, seq)$ message to $v$. □

Armed with Lemma 4.23, we are ready to prove Lemma 4.22:

*Proof.* Assume there is an admissible state $S$ and a node $u$ that has a SEARCH$(u, destID)$ message $m$ buffered at $u$ in $S$. Furthermore, let $seq$ be the value of $u.seq[destID]$ in $S$. Node $u$ initiates a PROBE$(u, destID, Next, seq)$ message every time it executes TIMEOUT. According to Lemma 4.23, $u$ will eventually receive a PROBESUCCESS$(destID, dest)$ or a PROBEFAIL$(destID, seq)$ message. Note that $u$ forwards or drops $m$ upon the first receipt of such message after $S$.

First, consider the case that the first such message that $u$ receives after state $S$ is a PROBESUCCESS$(destID, dest)$ message. Invariant 5 (which holds due to

Lemma 4.19) yields $id(dest) = destID$ and $m$ will be sent to $dest$, according to the protocol.

Second, consider that the first such message that $u$ receives is a ProbeFail$(destID, seq)$ message. According to Invariant 6, either no node $v$ with $id(v) = destID$ exists (in which case we are finished) or for every admissible state with $u.seq[destID] < seq, v \notin R(u, destID)$. Now consider an arbitrary earlier Search$(u, destID)$ message $m'$ that was buffered at $u$ during an admissible state. If $m'$ is still waiting at $u$ in state $S$, then $m'$ will be dropped together with $m$ when $u$ receives the ProbeFail$(destID, seq)$ message. Otherwise, assume for contradiction that immediately after some state $S' < S$, $m'$ was sent to a node $dest$ with $id(dest) = destID$. Since $m'$ was buffered at $u$ during at least one admissible state, by Lemma 4.19, $S'$ is admissible as well. According to the protocol, the fact that $m'$ was sent to $dest$ requires that there was a ProbeSuccess$(destID, dest)$ message in $u.Ch$ in $S'$. By Invariant 5, $dest \in R(u, destID)$ held in $S'$. Additionally, $u$ increased $u.seq[destID]$ when the first Search() message was initiated after that state, i.e., before $S$. Since the sequence numbers are monotonically increasing, $S'$ is a state with $u.seq[destID] < seq$. Thus, Invariant 6 of the ProbeFail$(destID, seq)$ message implies $dest \notin R(u, destID)$ in state $S'$, yielding a contradiction. This finishes the proof. □

Last, and building on Lemma 4.22, we are able to prove the following lemma:

**Lemma 4.24.** *In every computation of an ms-sufficient protocol, for every two nodes $u$ and $v$ such that $v \in R(u, id(v))$ in some admissible state $S$, there is a state $S' \geq S$ such that all* Search*$(u, id(v))$ messages initiated in $S'$ and all subsequent states will be delivered to $v$.*

*Proof.* Assume $v \in R(u, id(v))$ in an admissible state $S$. Note that according to Lemma 4.22, there will be a state $S' \geq S$ in which $u.WaitingFor[id(v)]$ is empty because all Search$(u, id(v))$ messages buffered at $u$ during $S$ have been sent to their destination or dropped right before (if there were any). Let $seq'$ be the value of $u.seq[id(v)]$ in $S'$. Consider the first Search$(u, id(v))$ message initiated after $S'$. Since InitiateNewSearch$(id(v))$ is the only action that adds elements to $u.WaitingFor[id(v)]$, $u.WaitingFor[id(v)] = \emptyset$ holds when this message is initiated. According to the pseudocode (c.f. Listing 4.2), $u.seq[id(v)]$ will be increased by one at that time: i.e., its new value will be $seq'' = seq + 1$. Now consider an arbitrary Search$(u, id(v))$ message $m$ initiated after $S'$. By Lemma 4.22, $m$ will be delivered or dropped. Assume for contradiction it is dropped. According to the pseudocode (c.f. Listing 4.2), this requires $u$ to receive a ProbeFail$(destID, seq)$ message with $destID = id(v)$ and $seq \geq u.seq[id(v)]$. Note that as argued before at that time $u.seq[id(v)] \geq seq'' > seq'$. Thus, Invariant 6 for the ProbeFail$(destID, seq)$ yields a contradiction to $v \in R(u, id(v))$ in $S$. All in all, we have that $m$ is delivered correctly. □

Using Lemma 4.19, Lemma 4.22 and Lemma 4.24, we can finally prove Theorem 4.17, which finishes this section. Before proving that theorem, we restate it as

follows:

**Theorem 4.17.** *Every ms-sufficient protocol satisfies monotonic searchability according to Invariants 1-7.*

*Proof.* Consider an arbitrary computation $C$ of an $ms$-sufficient protocol. According to Lemma 4.19 $C$ contains an admissible state and the computation suffix starting from the first admissible state $S$ solely consists of admissible states. Thus it remains to prove that the protocol satisfies monotonic searchability in this computation suffix.

Consider an arbitrary SEARCH$(u, destID)$ message $m$ initiated in a node $u$ after $S$ that is successfully delivered to the node $v$ with $id(v) = destID$. Assume for contradiction that there is another SEARCH$(u, destID)$ message $m'$ initiated after $m$ which is dropped. Note that both $m$ and $m'$ were buffered at $u$ during at least one admissible state. Lemma 4.22 implies that since $m'$ is dropped, $m$ must have been dropped as well, which represents a contradiction. Thus, every SEARCH$(u, destID)$ initiated after $m$ is delivered as well.

Last, note that by Definition 4.1 in legitimate states (i.e., in the target topology) for every pair of nodes $u$ and $v$, $v \in R(u, id(v))$. Thus, Lemma 4.24 yields that the non-triviality property is also fulfilled, which completes the proof of Theorem 4.17. □

## 4.5. Examples

In this section, we give some examples of topologies for which a self-stabilizing protocol that satisfies non-trivial monotonic searchability according to the generic search protocol can be obtained by combining existing results in the literature with the results presented in this chapter.[1] By Corollary 4.16 and Theorem 4.17, all that needs to be shown is that the target topology $T$ is feasible, that there is a fast search protocol $S_T$ and that the self-stabilizing protocol for the target topology fulfills the MCP.

### 4.5.1. The Line Graph

The line topology is the first topology for which a protocol satisfying non-trivial monotonic searchability was shown [SSS15]. Now, using the generic approach described in this chapter and applying it to the self-stabilizing list protocol presented by Nor, Nesterenko, and Scheideler (called $l$-Corona in [NNS13]), the non-trivial monotonic searchability follows as a corollary.

First of all, it is obvious that the linear list is feasible and that there is a fast routing protocol (which simply forwards the message into the direction of the target at each step). Second, note that in the protocol in [NNS13] each node has at most

[1] Note that some of the results we cite have been obtained for a synchronous model. However, the results easily transfer to the asynchronous model.

one neighbor in each direction and whenever a node $u$ receives a new reference $v$, $u$ keeps the reference if and only if it is closer to its previous neighbor $w$ in the same direction. In any case, $u$ delegates the non-kept reference to the kept neighbor in the same direction. Although $l$-Corona does not have a dedicated DELEGATE() message type, there is exactly one type of messages whose single parameter is a node reference and whenever a reference is removed from a node's variables, it is delegated in this type of messages, which thus satisfies the requirements of the DELEGATE() message type in Definition 4.6. Thus, this protocol fulfills the MCP.

### 4.5.2. The SKIP$^+$ Graph

The $SKIP^+$ graph was introduced by Jacob, Richa, Scheideler, Schmid, and Täubig [Jac+14] as a supergraph of the skip graph to overcome the issue that the consistency of the original skip graph [AS07] is not locally checkable. We now briefly recap the formal definition of a $SKIP^+$ graph from [Jac+14] and its self-stabilizing protocol and then show how to apply the results of this chapter to that topology.

#### SKIP$^+$ Graph Formal Definition

Each node $v$ has an identifier $v.id$ and a bit string $v.rs$ associated with it. Both $v.id$ and $v.rs$ together make up the identity of a node. $v.id$ is unique and otherwise chosen arbitrarily and there exists a total order on all these values. $v.rs$ is a random bit string in which each bit is chosen independently and uniformly at random. Although there is no specific bound on the length of these bit strings, they only need to be "sufficiently long", so that the bit strings of every two nodes differ from each other. If the required length is unknown in advance, the bit strings could be appended on demand. Note that in practical applications, the random bit strings can be computed with the help of a pseudo-random hash function from the $id$ value. We also denote the length-$i$-prefix of $v.rs$ by $pre_i(v)$ ($pre_0(v)$ is an empty bit string for every $v$ then). Let $\bot.id := -\infty$ and $\top.id := \infty$. [Jac+14] then defines for every node $v$, every subset $W$ of nodes, every $i \geq 0$, and every $x \in \{0,1\}$ ($\circ$ means string concatenation):

$$\begin{aligned} pred(v,W) &= \text{argmax}_{w \in W \cup \{\bot\}}\{w.id < v.id\}, \\ succ(v,W) &= \text{argmin}_{w \in W \cup \{\top\}}\{w.id > v.id\}, \\ pred_i(v,x) &= pred(v, \{w \in V | pre_{i+1}(w) = pre_i(v) \circ x\}), \\ succ_i(v,x) &= succ(v, \{w \in V | pre_{i+1}(w) = pre_i(v) \circ x\}). \end{aligned}$$

For every node $v$ and every $i \geq 0$, $range_i(v)$ is defined as:

$$range_i(v) = [\min\{pred_i(v,0).id, pred_i(v,1).id\}, \max\{succ_i(v,0).id, succ_i(v,1).id\}]$$

Intuitively, ranges at level $i$ are chosen such that they contain at least one level-$(i+1)$-predecessor with last bit 0 and at least one with last bit 1 as well as at least

one level-$(i+1)$-successor with last bit 0 and at least one with last bit 1. The *neighborhood at level i* is defined as $N_i(v) := \{w \in V | pre_i(w) = pre_i(v) \wedge w.id \in range_i(v)\}$. The neighborhood of a node $v$ is defined as the union $N_i(v)$ for all $i \geq 0$.

### Build-SKIP$^+$

The self-stabilizing protocol for the $SKIP^+$ graph described in [Jac+14] is called $Build{-}SKIP^+$ in the following. We define two types of edges: stable edges and temporary edges. An edge $(u, v)$ is called *stable* if $v.id \in range_i(u)$ for some $i$. Otherwise, $(u, v)$ is called *temporary*. The level of a temporary edge $(u, v)$ is defined as the length of the longest common prefix of $u$ and $v$. In the $Build{-}SKIP^+$ protocol, every node regularly informs all neighbors about itself (in order to establish reverse edges), determines its stable edges and additionally performs the following four rules ($N^l(u)$ is used to distinguish the *local neighborhood* of node $u$ during some state from the ideal neighborhood in the final topology):

1. Range Reduction. Every node $u$ with a stable neighbor $v$, for every $i \geq 0$ and every (not necessarily stable) neighbor $w \in N^l(u)$ such that $w \neq v$, $pre_i(v) = pre_i(w)$ and $w.id \in range_i(v)$, introduces $w$ to $v$. Additionally, if $v.id \in range_i(w)$, $v$ also introduces $v$ to $w$.

2. Forward Edges. Every node $u$ delegates every temporary edge $(u, v)$ to the stable neighbor $w$ of $u$ that has the largest common prefix with $v.rs$.

3. Local Closure. Each node $u \in V$ for all its neighbors $v, w \in N^l(u)$ introduces $w$ to $v$ if and only if its stable neighborhood changes in the following sense: In contrast to the previous round, either at least one stable edge $e = (u, x)$, for some $x \in V$ became instable, or the lowest level on which $e$ is stable changed, or $u$ is incident to a new stable edge.

4. Linearize. For every prefix length $i$, every node $u$ identifies the stable neighbors $v_1, \ldots, v_k$ with $v_1.id < v_2.id < \cdots < v_k.id$ having the prefix $pre_i(u) \circ 0$ and introduces $v_2$ to $v_1$, $v_3$ to $v_2$, ..., as well as $v_k$ to $v_{k-1}$. The analogous is applied to the stable neighbors $w_1, \ldots, w_l$ with $w_1.id < w_2.id < \cdots < w_l.id$ having the prefix $pre_i(u) \circ 1$.

Examples and intuitive descriptions as well as the full analysis of this protocol are provided in [Jac+14].

### Application of Our Results to the SKIP$^+$ Graph

Obviously, the $SKIP^+$ graph is a feasible topology and the search strategy that always forwards a message via a stable edge that "fixes one bit" of the destination is a fast search protocol. Thus, all that remains to be shown is that $Build{-}SKIP^+$ fulfills the MCP with respect to this search protocol. First of all, regarding the first

sub-property of the MCP, note that there is one (set) variable $N_i$ for each level $i$ storing the neighborhood at level $i$. Define $\gamma^{min}_{u,N_i}(A_{N_i})$ as the smallest element in $range_i(v)$ given the assignment $A_{N_i}$ and, accordingly, $\gamma^{max}_{u,N_i}(A_{N_i})$ as the greatest element in $range_i(u)$. Note in the above protocol description that a node $u$ never adds a node $v$ to its stable neighborhood at level $i$ if $u.id \notin range_i(u)$. Furthermore, a node $u$ removes a node $v$ from its stable neighborhood at level $i$ only if $range_i(v)$ changed in the same action. By the definition of $range_i(v)$ and the aforementioned, the lower end of this range can only increase and the upper end of this range can only decrease. Thus, the first sub-property of the MCP holds for $Build{-}SKIP^+$. For the second property, consider the case in which an edge $(u, v)$ is delegated to some other node $w$. This only happens in the second rule of $Build{-}SKIP^+$. In this case, $(u, v)$ is a temporary edge, $(u, w)$ is a stable edge, and $w$ has the largest common prefix with $v$. This implies that $\min(u.id, v.id) < w.id < \max(u.id, v.id)$. Thus, the delegation in that step satisfies the requirements of the DELEGATE() message type. All in all, $Build{-}SKIP^+$ fulfills the MCP.

[Jac+14] also argues that routing between any pair of nodes using the above fast search protocol is possible in time $O(\log n)$ with high probability. Thus, Corollary 4.18 implies the following:

**Corollary 4.25.** *In a stabilized $SKIP^+$ graph consisting of $n$ nodes, successful search requests are answered in time $O(\log n)$ with high probability.*

### 4.5.3. The Linearized De Bruijn Network

The linearized De Bruijn network (LDB) introduced by Richa, Scheideler, and Stevens [RSS11] is a discretization of a continuous variant of the well-known De Bruijn graph, in which the identifiers of the $n = 2^d$ nodes are bit strings of length $d$ and each node $u$ is connected to the nodes whose identifiers are obtained by shifting $u$'s identifier by one into either direction and padding the remaining bit with either 1 or 0. The following definition of the LDB is taken from [RSS11]:

**Definition 4.26** (The Linearized De Bruijn Network (LDB) [RSS11])**.** *$G = (V, E)$ is a directed graph where the node set $V$ can be partitioned into the set of* real *nodes $V_R$ and a set of* virtual *nodes $V_V$. Each real node $v \in V_R$ has a real-valued label in the interval $(0, 1)$. In addition, each $v \in V_R$ hosts two virtual nodes in $V_V$: a* left *virtual node, $l(v)$, with label $\frac{v}{2}$ and a* right *virtual node, $r(v)$, with label $\frac{v+1}{2}$. The collection of all real and virtual nodes $v \in V$ is arranged in sorted order of their labels and $(v, w) \in E$ if and only if $v$ and $w$ are consecutive in the linear ordering (linear edges) or $w$ is a virtual node of $v$ (virtual edges).*

It is obvious that the LDB is a feasible topology and that there is a fast search protocol $S_T$ (here one can use the same as for the line graph). Unfortunately, the routing protocol from [RSS11] that has a much better running time does not satisfy the definition of a fast routing protocol, since a single message is possibly routed into both the left and the right direction in the course of the routing. An important

property of the self-stabilizing algorithm for the LDB presented in [RSS11] is that each node $v \in V$ has at most one left and at most one right neighbor and that whenever such a neighbor is removed, it is replaced by a closer one. Furthermore, the previous neighbor is delegated to the new one and although the linearization is not described explicitly in [RSS11], it is not difficult to implement the protocol such that the required DELEGATE() message type is used for this.

Of course, one might question the usefulness of the De Bruijn graph if searching is done without exploiting the list edges. However, one might think of scenarios in which for part of the messages, monotonic searchability is required, whereas for the other messages, delivery speed is more important. One would thus be able to trade speed for reliability by sending the messages via the above protocol instead of via the standard De Bruijn search protocol.

## 4.6. A Short Digression: The Bridge-SKIP$^+$ Graph

We now consider a special approach for the $SKIP^+$ graph that allows for efficient monotonic searchability without using the framework introduced in this chapter. The result is a simpler protocol, although the resulting graph is only a supergraph of the $SKIP^+$ graph, called the $Bridge{-}SKIP^+$ graph. The idea of the self-stabilizing protocol $Build{-}Bridge{-}SKIP^+$ for the $Bridge{-}SKIP^+$ graph is the following: Each node $u$ for every level $i \geq 0$ and every $b \in \{0, 1\}$ maintains a *search neighbor* $s_{i,b}(u)$ that it keeps additionally to its (temporary and stable) neighborhood. Simply put, a node $v$ becomes $s_{i,b}(u)$ when the following two conditions are fulfilled: (i) $u$ does not yet have a search neighbor $s_{i,b}(u)$, and (ii) $v$ is $u$'s closest neighbor such that $prefix_{i+1}(v) = prefix_i(u) \circ b$. Once a node has become $s_{i,b}(u)$, the value of $s_{i,b}(u)$ remains unchanged. The search protocol then resembles the bit adaptation strategy with respect to the search edges only. We assume that nodes search for other nodes via their random bit string. The search protocol then works as follows[2] ($s_{i,b}(u) = \perp$ means that $u$ does not have a search neighbor for level $i$ and bit $b$, and $destRS[i+1]$ is the $(i+1)$-th bit of $destRS$):

Listing 4.3: $Bridge{-}SKIP^+{-}Search$

```
1  INITIATENEWSEARCH(destRS)
2     create a new message m = SEARCH(self, destRS, 0)
3     send m to self
4
5  SEARCH(u, destRS, i)
6     if self.id = destRS then
7     |  // success
8     else if s_{i,destRS[i+1]}(self) ≠ ⊥ then
9     |  send SEARCH(u, destRS, i + 1) to s_{i,destRS[i+1]}(self)
10    else
11    |  // fail, inform u about the failed message if desired
```

[2] In contrast to the search protocol from Section 4.4, the SEARCH() message type has a third parameter here, which allows us to keep the pseudocode short.

We obtain the following theorem:

**Theorem 4.27.** *$Build{-}Bridge{-}SKIP^+$ satisfies monotonic searchability according to $Bridge{-}SKIP^+{-}Search$.*

*Proof.* Note that the fact that $s_{i,b}(u)$ for every node $u$ and every $i$ and $b$ changes only when its previous value was $\perp$ directly implies the monotonicity property of monotonic searchability (each successful request from node $u$ to node $v$ takes exactly the same path in every search). Thus, all that needs to still be shown is the non-triviality property of $Build{-}Bridge{-}SKIP^+$ according to $Bridge{-}SKIP^+{-}Search$.

Therefore, consider an arbitrary search SEARCH$(u, destRS, i)$ request initiated *after* the underlying $SKIP^+$ graph has stabilized (which it does eventually as proven in [Jac+14]) and assume this request is dropped in some node $v$ (i.e., it reaches Line 11) although a node with bit string $destRS$ exists. Let $i$ be the value of the third parameter of SEARCH() when the message is dropped. By induction we obtain that $pre_i(v)$ equals the first $i$ bits of $destRS$. Furthermore, since the message is dropped, $s_{i,b} = \perp$, where $b$ is the $(i+1)$-th bit of $destRS$. This implies that $v$ does not have a neighbor $w$ such that $pre_{i+1}(w) = pre_i(v) \circ b$. Since we assumed a node with bit string $destRS$ to exist, there is at least one node $w'$ such that $pre_{i+1}(w') = pre_i(v) \circ b$ in the graph. According to the definition of a $SKIP^+$ graph (see Section 4.5.2), the closest such node $w''$ is in $range_i(v)$ and thus in $N(v)$. Therefore, $w''$ would be a search neighbor of $v$ for level $i$ and bit $b$, which yields a contradiction. This completes the proof of the lemma. □

Note that $Bridge{-}SKIP^+{-}Search$ fixes one bit in every step and the maximum required size of the bit strings is $O(\log n)$ with high probability (see [Jac+14]). Furthermore, each node stores at most one search neighbor for every *non-trivial* level $i$, i.e., for every level in which there possibly is a neighbor. It follows from Chernoff bounds that the number of these levels is $O(\log n)$ with high probability (see, [Jac+14]). Thus, altogether we obtain the following result:

**Corollary 4.28.** *$Build{-}Bridge{-}SKIP^+$ satisfies monotonic searchability according to $Bridge{-}SKIP^+{-}Search$ and all search messages are delivered or dropped within $O(\log n)$ steps with high probability. The number of additional edges of a $Bridge{-}SKIP^+$ graph (in comparison to the $SKIP^+$ graph consisting of the same nodes) is $O(\log n)$ per node with high probability (i.e., the total number of edges increases by a factor of $O(1)$ w.h.p.).*

CHAPTER 5

# Monotonic Searchability under Leaving Nodes

So far, we have considered monotonic searchability in self-stabilizing systems assuming that the node set is static. Yet if we also allow nodes to leave the system, the problem becomes much more challenging. In this chapter we thus consider the problem of monotonic searchability in systems with leaving nodes. To reduce the complexity, we do not provide a generic solution to this problem here. Instead, as a starting point, we target the nodes to form the line topology and we will see that even for a topology as simple as this, the solution will be quite complex.

The main results of this chapter have previously appeared in the following publication:

> Christian Scheideler, Alexander Setzer, and Thim Strothmann. **Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures**. In: *Proceedings of the 19th International Conference on Principles of Distributed Systems (OPODIS)*. Rennes, France, 2015. [SSS15]

**Outline of This Chapter** The outline of this chapter is as follows: First, in Section 5.1, we formally define the problem of monotonic searchability under leaving nodes. Then, in Section 5.2, we present the Build-List* protocol and the corresponding search protocol, Search*. In the subsequent sections, we further show that Build-List* solves the $\mathcal{FDP}$ (Section 5.3), self-stabilizes to the line topology (Section 5.4) and thereby satisfies monotonic searchability (Section 5.5).

## 5.1. Problem Statement

This chapter considers the same communication model as Chapter 4 (the description of this model is provided in Section 4.1.1). We consider the problem of forming a line topology, solving the $\mathcal{FDP}$, and satisfying monotonic searchability. For this problem, we can apply the same definition of monotonic searchability as in Chapter 4, with the only difference that monotonic searchability only needs to hold for pairs of staying nodes. For completeness, we state the adapted definition here again:

**Definition 5.1** (Monotonic Searchability under leaving nodes)**.** *A self-stabilizing protocol $P$ satisfies* monotonic searchability *according to some search protocol $P_S$ if search requests are routed according to $P_S$ and it fulfills the following two properties for every computation $C$ of $P$:*

***Monotonicity*** *For every pair of nodes $v, w$ that remain staying throughout $C$ and every* SEARCH*$(v, id(w))$ request $r$ initiated in some state $S$ of $C$, every* SEARCH*$(v, id(w))$ request $r'$ such that $r'$ is initiated in a state $S' \geq S$ also succeeds.*

***Non-Triviality*** *$C$ has a suffix such that in this suffix for every pair of nodes $v, w$,* SEARCH*$(v, id(w))$ requests will succeed if $v$ and $w$ remain staying throughout $C$ and there is a path from $v$ to $w$ in the target topology.*

Analogous to the notation used in Chapter 4, for three nodes $u, v, w$ we say $v$ is *closer* to $u$ than $w$ if and only if $|id(v) - id(u)| < |id(w) - id(u)|$.

In [For+14] it was shown that in the model considered in this chapter, there is no distributed protocol that can decide when it is safe for a node $u$ to leave the system, i.e., whether $u$'s executing **exit** would disconnect the network graph. Therefore, solving the $\mathcal{FDP}$ is impossible without any further additions to the model. The authors solve the issue by introducing an oracle. In general, an *oracle* is a predicate that depends on the current system state and the node calling it. In the context of the $\mathcal{FDP}$, an oracle is supposed to advise a leaving node when it is safe to execute **exit**. We use the oracle $\mathcal{NIDEC}$ as introduced in [For+14] in order to solve the $\mathcal{FDP}$:

**Definition 5.2** ($\mathcal{NIDEC}$ oracle)**.** *For an arbitrary node $u$ calling the oracle $\mathcal{NIDEC}$, $\mathcal{NIDEC}$ evaluates to* ***true*** *if and only if:*

1. *no node $v \neq u$ has a reference to $u$ in its local memory or in a message in $v.Ch$ and*
2. *$u.Ch$ is empty.*

We note that the $\mathcal{FDP}$ deliberately ignores that new nodes can join the network. However, this abstraction is justified in a self-stabilizing setting, since from an algorithmic point of view for some node $u$ a new node joining the network is the same as getting a message from a node that it has never been in contact with.

To sum up, the goal of this chapter is to obtain a protocol that stabilizes to the list, solves the finite departure problem and satisfies monotonic searchability according to a corresponding search protocol.

## 5.2. Protocol Description of Build-List* and Search*

In this section, we describe the self-stabilizing BUILD-LIST* protocol and the corresponding search protocol SEARCH*. To simplify the description, we first describe the protocol in such a way that we assume only staying nodes to exist in the system (Section 5.2.1) and then describe the differences that apply when there are leaving nodes as well (Section 5.2.2). In doing so, we first restrict ourselves to describing only the parts of the protocol that are relevant for the topology stabilization. Afterwards, we deal with the parts that belong to the search

protocol (Section 5.2.3). At the end of this section, we give the full specification of Build-List* and Search* (Section 5.2.4).

### 5.2.1. Protocol Overview Assuming Staying Nodes Only

The general idea used to make convergence to the line topology possible is the linearization technique introduced in [ORS07]. In a nutshell, it means that each node always keeps the closest neighbor in each direction and delegates all neighbors it does not need. However, as in Chapter 4, we need to apply some kind of "Safe Delegation" technique because normal delegations could replace explicit edges by paths of explicit and implicit edges preventing search requests from reaching the target.

Thus, in our protocol, each node $u$ has sets of neighbors $Left$ and $Right$ used to store neighbors with smaller or greater identifier, respectively. In the following, for a node $u$ we refer to these sets with the notation $Left(u)$ and $Right(u)$. An important property of our protocol is that every node $w$ does not delegate any edge whose endpoint $v$ is stored in $Left(w)$ or $Right(w)$ directly. Instead it first introduces $v$ to another node $u$ via an Introduce$(v, w)$ message, waits for an acknowledgement that $v$'s reference has been added to $Left(u)$ or $Right(u)$ (which the Linearize$(v)$ message type is used for) and then delegates $v$ to a neighbor closer to $v$ (which it uses the SingleIntroduce$(v)$ message type for). More specifically, whenever a node $u$ has multiple right neighbors, it does the following (the behavior for the left neighbors is completely analogous): Let $w_1, \ldots, w_\ell$ be $u$'s right neighbors with $id(w_i) < id(w_{i+1})$ for all $1 \leq i < \ell$. In the Timeout action $u$ introduces $w_{i+1}$ to $w_i$ by sending an Introduce$(w_{i+1}, u)$ message to $w_i$. The second parameter enables $w_i$ to know that it received $w_{i+1}$'s reference from $u$. $w_i$ thus stores $w_{i+1}$'s reference (in $Right(w_i)$), sends a Linearize$(w_{i+1})$ message back to $u$ (to acknowledge the receipt of $w_{i+1}$'s reference) and sends a SingleIntroduce$(u)$ message to itself (the latter is only to preserve connectivity in case the message originates from an initial state). Node $u$ now acts upon that Linearize$(w_{i+1})$ message by deleting $w_{i+1}$ from its memory. Since references may never be thrown away, to preserve connectivity $u$ sends $w_{i+1}$'s reference to a right neighbor (in particular, the one with the maximum identifier whose identifier is still smaller than $w_{i+1}$'s — observe that this is not necessarily $w_i$ anymore). Since $u$ only removes the explicit edge to $w_{i+1}$ after it has received the acknowledgement from $w_i$, $u$ preserves a path of explicit edges between $u$ and $w_{i+1}$. The other thing $u$ does in the Timeout action is to send its own reference to the closest neighbors on each side via a SingleIntroduce$(u)$ message. Aside from this purpose, the SingleIntroduce$(u)$ action is basically used to delegate a reference of a node $u$ into one direction (i.e., to the left or to the right) as long as there is a node between the current node and $u$ in $Left$ or $Right$. Note that implicit edges are not used for searches, which is why we do not have to apply the "Safe Delegation" principle for this kind of edges. However, they still need to be delegated (since references may not be thrown away if connectivity is to be maintained). Further note that

even though a node may temporarily have more references than required for the line topology, BUILD-LIST* still eventually stabilizes to the line, as superfluous edges are constantly linearized.

### 5.2.2. Additional Details for Dealing with Leaving Nodes

After this description of the behavior of BUILD-LIST* in a system consisting of staying nodes only, we now describe the additional behavior of BUILD-LIST* also under the presence of leaving nodes. In contrast to the staying nodes, each leaving node distinguishes between two different kinds of neighbors: those that it already had before switching to the leaving mode (which are $Left$ and $Right$) and those which it received while being leaving ($Temp_L$ and $Temp_R$).

For the INTRODUCE(), LINEARIZE() and SINGLEINTRODUCE() actions, as well as the FORWARDPROBE() action, which we will specify for the search protocol later on, a leaving node $u$ will always save nodes in $Temp_L$ or $Temp_R$ in cases where a staying node saves them in $Left$ or $Right$, respectively. In its TIMEOUT action, a leaving node $u$ either introduces all its neighbors to each other and executes **exit** if $\mathcal{NIDEC}$ is true or it sends REVERSEANDLINEARIZEREQ() messages to all its neighbors. With these REVERSEANDLINEARIZEREQ() messages $u$ requests all neighbors to stop holding its reference. It is not advisable for leaving nodes to send their own reference to other nodes. Therefore, a REVERSEANDLINEARIZEREQ($dir$) message only contains a value $dir \in \{left, right\}$ that indicates whether a left or right neighbor should be removed: i.e., $u$ sends a REVERSEANDLINEARIZEREQ($left$) message to all its neighbors to the right and a REVERSEANDLINEARIZEREQ($right$) message to all its neighbors to the left. If a node $v$ receives a REVERSEANDLINEARIZEREQ($dir$) message, there are two possible scenarios. If $v$ is staying or $dir = right$, it sends a REVERSEANDLINEARIZEACK($v, uniqueValue$) message to all neighbors in the given direction, which contains its own reference and a uniquely created value for each neighbor (which can be implemented as easily as by a local counter). These values are also stored by $v$ at the corresponding node reference in the neighbor set. If, however, $v$ is leaving and $dir = left$, $v$ simply ignores the request to reverse its edge. Thereby, leaving nodes with a higher identifier are given a higher priority for exiting the system. Once a leaving node $u$ receives a REVERSEANDLINEARIZEACK($v, uniqueValue$) message, it responds with a REVERSEANDLINEARIZE($nodeList, uniqueValue$) message that contains the received unique value (for identification purposes) and also all its neighbors that are on the opposite of the node in the message (i.e., if the received node is to the right of $u$, $u$ sends all left neighbors and vice versa). A REVERSEANDLINEARIZEACK($v, uniqueValue$) message is ignored by a staying node, meaning that it is transformed into a SINGLEINTRODUCE($v$) message to itself. Finally, the REVERSEANDLINEARIZE($nodeList, uniqueValue$) message is received by $v$ and $v$ checks if it has a neighbor $u$ with the given unique value. If this is the case, $v$ either finishes the reversal process by reversing the edge $(v, u)$ and saving the newly received neighbors (if $v$ is staying

or getting the REVERSEANDLINEARIZE($nodeList, uniqueValue$) message from a right neighbor) or $v$ ignores the message by simply saving all nodes in $Temp_L$ (if $v$ is leaving and getting the REVERSEANDLINEARIZE($nodeList, uniqueValue$) message from a left neighbor). In case the unique value does not match, the REVERSEANDLINEARIZE($nodeList, uniqueValue$) message is not a response to a former REVERSEANDLINEARIZEACK($v, uniqueValue$) message (i.e., it is a corrupted message originating from the initial state) and thus $v$ only sends a SINGLEINTRODUCE($w$) message to itself for every received node $w \in NodeList$ in order to maintain connectivity.

### 5.2.3. The Search Protocol Search*

After having described the parts of the protocol relevant for topology stabilization, we now describe the search protocol SEARCH*. In general, this protocol uses the same probing approach used by the generic search protocol in Chapter 4. Even more, for staying nodes it behaves completely analogously. Whenever a staying node $u$ executes the INITIATENEWSEARCH($destID$) action (leaving nodes will do nothing during that action), $u$ creates a new SEARCH($u, destID$) message and starts to periodically initiate FORWARDPROBE($u, destID, \{u\}, u.lseq$) messages that it sends to itself. Each FORWARDPROBE($source, destID, Next, seq$) message contains four parameters, whose meaning is as follows: $source$ is the reference of the initiator of the message. $destID$ is the identifier of the target being search for. $Next$ is a set of node references containing nodes the message (or, to be more precise, a message caused by this message) will visit in the future. $seq$ is a counter used to distinguish different probing batches, which we will explain more about later on. Whenever a FORWARDPROBE($u, destID, Next, seq$) message is at a node $w$, $w$ removes itself from $Next$ and adds all its neighbors from $Left(w)$ or $Right(w)$ whose identifier is "closer" to $destID$ than $w$'s but still between $w$'s identifier and $destID$. Then it forwards the FORWARDPROBE($u, destID, Next, seq$) message to the node in $Next$ whose identifier has the greatest distance to $destID$. If a FORWARDPROBE($u, destID, Next, seq$) message arrives at a staying node $v$ with $id(v) = destID$, it directly responds with a PROBESUCCESS($destID, v$) message to $u$. Leaving nodes will not send such a message as they do not want to increase the number of nodes holding their references and because delivering a search request to these nodes is not necessary according to the problem definition anyway. If, however, $Next$ is empty at a node $w$ with $id(w) \neq destID$ after $w$ has added all neighbors to $Next$ according to the aforementioned rule (which were none in this case), the FORWARDPROBE() message is answered with a PROBEFAIL($destID, seq$) message. In any case, as soon as $u$ receives the response, it acts accordingly: If the answer to a FORWARDPROBE($u, destID, Next, seq$) message is a PROBEFAIL($destID, seq$) message, it drops the corresponding SEARCH($u, destID$) message completely (in fact, there is an additional detail depending on the value of $seq$, but this will be explained in the next paragraph). If the answer is PROBESUCCESS($destID, v$), SEARCH($u, destID$) messages waiting at $u$ are directly sent to $v$. In addition to

the aforementioned, since nodes must not throw away node references, in all of these functions, nodes will deliberately store or delegate references retrieved via parameters of messages. Here, staying nodes might even add additional nodes to $Left$ and $Right$ if they would be closer neighbors than the previous ones on the "same side". Leaving nodes, however, will add these nodes only to $Temp_L$ and $Temp_R$.

To implement the aforementioned probing mechanism, if additional SEARCH($u$, $destID$) messages are created at $u$ while $u$ is still waiting for an answer to a previously initiated FORWARDPROBE($u, destID$) message, these requests are buffered at $u$ (for which the $WaitingFor[destID]$ map is used) and are dropped or delivered as soon as the PROBEFAIL($destID, seq$) or PROBESUCCESS($destID, v$) response arrives at $u$. This means that search requests to the same destination might be sent out in batches. Furthermore, note that nodes do not memorize whether they have already sent FORWARDPROBE() messages to a certain destination. Due to corrupted initial states, this knowledge could be wrong and nodes relying on this knowledge would wait forever. Therefore, nodes periodically send FORWARDPROBE() messages during TIMEOUT, and not only once. We now explain the purpose of the fourth parameter $seq$ of the FORWARDPROBE() message. Recall that we make no assumptions on the message delivery speed and that channels do not guarantee FIFO delivery. Therefore, it is possible that a node $u$ receives a PROBEFAIL() message that is actually an answer to a FORWARDPROBE() message initiated long ago. Even worse, there might have been successful responses in the meantime, which caused $u$ to deliver search messages to the destination. Since dropping messages upon receipt of this PROBEFAIL() message now would certainly contradict the definition of monotonic searchability, each node $u$ stores a sequence number counter $lseq$. Whenever INITIATENEWSEARCH($destID$) is executed by $u$ and there is no SEARCH($u, destID$) that waits for an answer to a FORWARDPROBE() message, $u$ increments $u.lseq$, stores the new $u.lseq$ value in an entry for $v$ and always attaches the current sequence number ($u.lseq$) to each FORWARDPROBE() message it sends. Negative responses to probes (i.e., PROBEFAIL() messages) also contain this sequence number to identify the batch this message belongs to. Whenever $u$ receives a PROBEFAIL() message, $u$ checks whether the sequence number in this message is at least the sequence number stored for $destID$. If not, it drops the message, since in that case, the answer belongs to a FORWARDPROBE() message sent for an earlier batch of SEARCH($u, destID$) messages that have already been processed.

### 5.2.4. The Pseudocode of Build-List* and Search*

To keep the pseudocode concise, there is one part of the protocol that is only relevant to dealing with initial states. We therefore describe this part in words: At the beginning of each action (before the actual code starts), BUILD-LIST* performs a sanity check for $Left$, $Right$, $Temp_L$, and $Temp_R$ and possibly reorders the references accordingly (such that $Left$ and $Temp_L$ store only references of nodes

with a smaller identifier than the executing node and $Right$ and $Temp_L$ store only references of nodes with a greater identifier). Additionally, it ensures that there are no two identifiers $u \neq v$ such that $uniqueValues[u] = uniqueValues[v]$ (otherwise it assigns a new unique value to one of the two). The effect of this maintenance is summarized by the following lemma (which is easily obtained by checking that in the pseudocode nodes add references to $Right$ or $Temp_R$ only if their identifier is greater than $self$ and to $Left$ or $Temp_R$ if the opposite is the case):

**Lemma 5.3.** *In every but the first state of every computation of* Build-List*, *for every node $v$ the following holds:* $\forall\, x \in Left \cup Temp_L : id(x) < id(v)$ *and* $\forall\, y \in Right \cup Temp_R : id(v) < id(y)$.

Listing 5.1: Build-List* protocol

```
1  Timeout
2    if mode = staying then
3    | for every destID such that WaitingFor[destID] ≠ ∅ do
4    | | send forwardProbe(self, destID, {self}, lseq) to self
5    | let Left = {v_1, v_2, ..., v_k} with id(v_k) < id(v_{k-1}) < ··· < id(v_1)
6    | for all v_i ∈ Left with 1 ≤ i < k
7    | | send Introduce(v_{i+1}, self) to v_i
8    | let Right = {w_1, w_2, ..., w_l} with id(w_1) < id(w_2) < ··· < id(w_l)
9    | for all w_i ∈ Right with 1 ≤ i < l do
10   | | send Introduce(w_{i+1}, self) to w_i
11   | send SingleIntroduce(self) to v_1
12   | send SingleIntroduce(self) to w_1
13   else // mode = leaving
14   | if NIDEC = true then
15   | | for all v ∈ Left ∪ Right ∪ Temp_L ∪ Temp_R do
16   | | | for all w ∈ Left ∪ Right ∪ Temp_L ∪ Temp_R do
17   | | | | send SingleIntroduce(w) to v
18   | | | | send SingleIntroduce(v) to w
19   | | exit
20   | else
21   | | for all v ∈ Left ∪ Temp_L do
22   | | | send ReverseAndLinearizeREQ(right) to v
23   | | for all w ∈ Right ∪ Temp_R do
24   | | | send ReverseAndLinearizeREQ(left) to w
25
26 Introduce(v, w)
27   if id(v) < id(self) then
28   | if mode = staying then
29   | | Left := Left ∪ {v}
30   | | send Linearize(v) to w
31   | | send SingleIntroduce(w) to self
32   | else // mode = leaving
33   | | if v ∉ Left then
34   | | | Temp_L := Temp_L ∪ {v}
35   | | if w ∉ Left then
36   | | | Temp_L := Temp_L ∪ {w}
37   else if id(v) > id(self) then
38   | // analogous to the previous case
```

```
39
40 LINEARIZE(v)
41    if id(v) < id(self) then
42    |  if mode = staying then
43    |  |  if Left ≠ ∅ then
44    |  |  |  x := argmax{id(x') : x' ∈ Left}
45    |  |  |  if id(v) > id(x) then
46    |  |  |  |  Left := Left ∪ {v}
47    |  |  |  if id(v) < id(x) then
48    |  |  |  |  w := argmin{id(w') : w' ∈ Left and id(w') > id(v)}
49    |  |  |  |  Left := Left \ {v}
50    |  |  |  |  send SINGLEINTRODUCE(v) to w
51    |  |  else // Left = ∅
52    |  |  |  Left := Left ∪ {v}
53    |  else // mode = leaving
54    |  |  Temp_L := Temp_L ∪ {v}
55    else if id(v) > id(self) then
56    |  // analogous to the previous case
57
58 SINGLEINTRODUCE(u)
59    if id(u) < id(self) then
60    |  if Left = ∅ then
61    |  |  if mode = staying then
62    |  |  |  Left := Left ∪ {u}
63    |  |  else
64    |  |  |  Temp_L := Temp_L ∪ {u}
65    |  else
66    |  |  x := argmax{id(x') : x' ∈ Left}
67    |  |  if id(x) < id(u) then
68    |  |  |  if mode = staying then
69    |  |  |  |  Left := Left ∪ {u}
70    |  |  |  else
71    |  |  |  |  Temp_L := Temp_L ∪ {u}
72    |  |  else
73    |  |  |  send SINGLEINTRODUCE(u) to x
74    else if id(u) > id(self) then
75    |  // analogous to the previous case
76
77 REVERSEANDLINEARIZEREQ(dir)
78    if dir = right then
79    |  for all v ∈ Right ∪ Temp_R do
80    |  |  if uniqueValues[v] = ⊥ then // i.e.: v does not exist in uniqueValues
81    |  |  |  // assume that generateUniqueValue() creates a unique value
82    |  |  |  uniqueValues[v] = generateUniqueValue()
83    |  |  send REVERSEANDLINEARIZEACK(self, uniqueValues[v]) to v
84    else if dir = left and mode = staying then
85    |  // analogous to the previous case
86
87 REVERSEANDLINEARIZEACK(v, uniqueValue)
88    if id(v) < id(self) then
89    |  if mode = leaving then
90    |  |  Temp_L := Temp_L ∪ {v}
91    |  |  send REVERSEANDLINEARIZE(Right, uniqueValue) to v
92    |  else
```

```
93      | | send SINGLEINTRODUCE(v) to self
94      else if (id(v) > id(self)) then
95      | // analogous to the previous case
96
97    REVERSEANDLINEARIZE(nodeList, uniqueValue)
98      if ∃ v ∈ Left ∪ Right ∪ Temp_L ∪ Temp_R such that uniqueValues[v] = uniqueValue and
99        (id(v) < id(self) and ∀ w ∈ nodeList : id(w) < id(v) or
100       id(v) > id(self) and ∀ w ∈ nodeList : id(w) > id(v)) then
101     | if mode = staying then
102     | | if id(v) < id(self) then
103     | | | Left := Left ∪ nodeList
104     | | | Left := Left \ {v}
105     | | | send SINGLEINTRODUCE(self) to v
106     | | else if id(v) > id(self) then
107     | | | // analogous to the previous case
108     | else // mode = leaving
109     | | if id(v) < id(self) then
110     | | | Temp_L := Temp_L ∪ nodeList
111     | | else // id(v) > id(self)
112     | | | if v ∈ Right then
113     | | | | Right := Right ∪ nodeList
114     | | | | Right := Right \ {v}
115     | | | else
116     | | | | Temp_R := Temp_R ∪ nodeList
117     | | | | Temp_R := Temp_R \ {v}
118     | | | send SINGLEINTRODUCE(self) to v
119     else
120     | for all u ∈ nodeList do
121     | | send SINGLEINTRODUCE(u) to self
```

Listing 5.2: SEARCH* protocol

```
1     INITIATENEWSEARCH(destID)
2       if mode = staying then
3       | create a new message m = SEARCH(self, destID)
4       | if WaitingFor[destID] = ∅ then
5       | | lseq := lseq + 1
6       | | lseq[destID] := lseq // lseq[destID] is assumed to be -1 if its current value is not a
7       | |                      // positive integer
8       | // Store the messages in WaitingFor
9       | WaitingFor[destID] := WaitingFor[destID] ∪ {m}
10
11    FORWARDPROBE(source, destID, Next, seq)
12      if destID = id(self) then
13      | if mode = staying then
14      | | if Next ≠ ∅ then
15      | | | for all u ∈ Next do
16      | | | | send SINGLEINTRODUCE(u) to self
17      | | send PROBESUCCESS(destID, self) to source
18      | | send SINGLEINTRODUCE(source) to self
19      | else
20      | | send PROBEFAIL(destID, seq) to source
21      | | for all u ∈ Next do
22      | | | send SINGLEINTRODUCE(u) to self
```

```
23      | | send SingleIntroduce(source) to self
24      else
25      | if destID > id(self) then
26      | | Next := Next \ {self} ∪ {w ∈ Right : id(w) ≤ destID}
27      | | if Next = ∅ then
28      | | | send ProbeFail(destID, seq) to source
29      | | | send SingleIntroduce(source) to self
30      | | else
31      | | | u := argmin{id(u) : u ∈ Next}
32      | | | if id(u) < id(self) then
33      | | | | send SingleIntroduce(u) to self
34      | | | else if id(u) < min{id(v) : v ∈ Right} then
35      | | | | if mode = staying then
36      | | | | | Right := Right ∪ {u}
37      | | | | else
38      | | | | | Temp_R := Temp_R ∪ {u}
39      | | | send ForwardProbe(source, destID, Next, seq) to u
40      | if destID < id(self) then
41      | | // analogous to the previous case
42
43  ProbeSuccess(destID, dest)
44      if mode = staying then
45      | send all m ∈ WaitingFor[destID] to dest
46      | WaitingFor[destID] := ∅
47      send SingleIntroduce(dest) to self
48
49  ProbeFail(destID, seq)
50      if mode = staying then
51      | if seq ≥ lseq[destID] then
52      | | // the message belongs to the current set of buffered search requests for destID
53      | | WaitingFor[destID] := ∅
```

## 5.3. Build-List* Solves the $\mathcal{FDP}$

In this section we prove that Build-List* solves the finite departure problem, which is formalized by the following theorem:

**Theorem 5.4.** Build-List* *is a self-stabilizing solution to the* $\mathcal{FDP}$.

The remainder of this section is dedicated to proving this theorem. First of all, we prove a property that is also called the *safety* property. Let $PNG$ be the subgraph of $NG$ induced by the active nodes.

**Lemma 5.5.** *If a computation of* Build-List* *starts in a state in which* $PNG$ *is weakly connected,* $PNG$ *remains weakly connected in every state of this computation.*

*Proof.* We prove the lemma by showing that none of the actions of the protocol disconnects $PNG$. First of all, note that whenever a node executes **exit**, $\mathcal{NIDEC}$ was true before. Thus, such an exiting of a node cannot disconnect $PNG$. Second,

check in the pseudocode that no action "throws away" a reference received as a parameter of the message and not already stored in one of the node's variables. Either the received reference is added to one of the node's variables, or a message containing that reference is sent to the node itself or a neighbor. Third, consider the only occasions at which a node $v$ is removed from one of the variables of a node $u$. This happens in LINEARIZE(), in which case $v$ is delegated to another neighbor of the executing node, or in REVERSEANDLINEARIZE(), in which case $u$ sends a reference of itself to $v$: i.e., it applies the reversal primitive on the edge $(u, v)$. Therefore, none of the actions of BUILD-LIST* disconnects $PNG$. □

The main part of this section will consist in proving the following property, which is also called the *liveness* property:

**Lemma 5.6.** *For any computation of* BUILD-LIST* *there exists a computation suffix in which all leaving nodes are inactive.*

We establish a sequence of other lemmas to prove this lemma. Before that, however, we describe the idea of the proof of Lemma 5.6: For simplicity, assume for a moment that the nodes already form a sorted list. A single leaving node $u$ whose left and right neighbors are staying can easily exit: $u$ simply asks its neighbors to reverse their connection to $u$ (meanwhile informing them about the neighbors in the opposite direction to "bridge" $u$) and waits until all these connections have been reversed. As soon as $\mathcal{NIDEC}$ is true, $u$ can exit. The problem becomes more difficult when there is a sequence of leaving nodes in the list: If the leaving neighbors of a leaving node $u$ neatly reversed their connections to $u$, they would send their own references to $u$. But if $u$ did the same to them, this would cause an infinite loop. The protocol tackles this issue by giving "priority" to the leaving nodes with a greater identifier: A leaving node does reverse its connection to a right leaving node, but not to a left leaving node. At first glance, it might seem that the rightmost leaving node in the system is the first one which may exit then. Unfortunately, things are not that easy: The rightmost leaving node $u$ in the system may have an incoming connection from a leaving node $v$ which it does not have a connection to. Thus, $u$ then cannot send a REVERSEANDLINEARIZEREQ() message to $v$. However, it is crucial that $v$ does not introduce itself to its neighbors as this would possibly prevent $\mathcal{NIDEC}$ from becoming true at all. Therefore, we apply a trick: We assume for contradiction that at some point in time, no more nodes will execute **exit**. It is not difficult to prove that there is always a state then such that the left neighborhoods of all leaving nodes are fixed. This insight enables us to reasonably define the "leftmost" leaving node $u^*$ that will not be the left neighbor of any leaving node in the remainder of the computation. The idea of choosing this node is that every leaving node "left of $u^*$" will regularly receive a REVERSEANDLINEARIZEREQ() and thus also send an according REVERSEANDLINEARIZEACK() message to $u^*$ (if they have $u^*$ as a right neighbor), allowing $u^*$ to ask these nodes to reverse their connection to $u^*$. This way, $u^*$ will eventually get rid of all incoming connections "from the left".

We can show that it will also get rid of all incoming connections "from the right", which eventually enables $u^*$ to leave, yielding the desired contradiction.

To simplify the notation we introduce the following convenient definitions:

**Definition 5.7** (Incoming / Outgoing Neighbor, Left / Right Neighbor)**.** *If in some state $S$ there is an edge $(u, v)$ then $u$ is called an* incoming neighbor *of $v$ in $S$. Furthermore, to avoid confusion, we may refer to $v$ as the* outgoing neighbor *of $u$.*

*An (incoming / outgoing) neighbor $v$ of $u$ is called a* right (incoming / outgoing) neighbor *if $id(v) > id(u)$, or a* left (incoming / outgoing) neighbor *if $id(v) < id(u)$. We refer to the set of left or right neighbors by the* left *or* right neighborhood*, respectively.*

In the following, we will consider an arbitrary but fixed computation $C$ and show that every leaving node will eventually exit. To do so, note that there is a state $S_1$ in $C$ such that: (i) all nodes that will ever decide to be leaving have already done so before $S_1$, and (ii) all leaving nodes that will eventually execute **exit** are inactive in $S_1$. As explained before, we will establish a contradiction to the assumption that there is an active leaving node after $S_1$.

We begin with the following lemma that summarizes some basic properties of Build-List*:

**Lemma 5.8.** *There is a state $S_2 \geq S_1$ such that in $SUFFIX(S_2)$:*

1. *the left neighborhood of every leaving node does not change,*
2. *there is no* ForwardProbe*$(source, destID, Next, seq)$ message such that source is leaving,*
3. *there is no* ProbeSuccess*$(destID, dest)$ message such that dest is leaving, and*
4. *if there is a* ForwardProbe*$(source, destID, Next, seq)$ message in $v.Ch$ then $destID \geq id(v)$ and for all $u \in Next$, $id(u) \geq id(v)$, or $destID \leq id(v)$ and for all $u \in Next$, $id(u) \leq id(v)$.*

*Proof.* For the first statement, note that a leaving node does not remove a left neighbor from its neighborhood unless it executes *exit*. Since we assume that no nodes do this after $S_1$, as soon as every leaving node has added every left neighbor it will ever add to its neighborhood, the left neighborhood of a leaving node does not change.

Furthermore, note that a leaving node never initiates a ForwardProbe() message. Whenever a ForwardProbe$(source, destID, Next, seq)$ message is sent, it is either initiated, in which case *source* is equal to the initiator, or it is sent upon receipt of a ForwardProbe$(source', destID', Next', seq')$ message, in which case $source = source'$. Note that the pseudocode of the ForwardProbe() action ensures that when a ForwardProbe() message is sent, only references of nodes

whose identifier is between that of the current node and $destID$ are added to $Next$. In the FORWARDPROBE() action, at most one new FORWARDPROBE() message is sent and when this happens it is sent to the node in $Next$ with the maximum distance to $destID$. By induction, this implies that each FORWARDPROBE($source, destID, Next, seq$) initiated at some node *causes* only a finite number of FORWARDPROBE() messages (i.e., they are sent upon receipt of that message or upon receipt of a message caused by that message). All in all, it can be seen that eventually there will be no FORWARDPROBE($source, destID, Next, seq$) message in the system such that *source* is leaving as such messages will not be initiated anymore.

For the third statement, notice that PROBESUCCESS($destID, dest$) is only sent by the node *dest*, a leaving node never does this.

For the fourth statement, recall that as we argued in the second paragraph of this proof, all FORWARDPROBE() messages initially in the system at the beginning of the computation will eventually be inactive in some state $S$. Thus, all we need to show is that the claim holds for every newly initiated FORWARDPROBE() message and every FORWARDPROBE() message caused by a FORWARDPROBE() message initiated by the protocol. For the former, note that every newly initiated FORWARDPROBE($source, destID, Next, seq$) message by a node $u$ is initiated with $Next = \{u\}$, thus the claim holds. For the latter, assume the claim holds for an existing FORWARDPROBE($source, destID, Next, seq$) message in $v.Ch$ for some node $v$. In the case that $destID \geq id(v)$, when $v$ sends a new FORWARDPROBE() message, it sends a FORWARDPROBE($source, destID, Next', seq$) message such that $Next' \subseteq Next \setminus \{u\} \cup Right(v)$ and sends it to the node $u$ with the minimum identifier in $Next'$, so the claim holds for the new message as well. In the case that $destID \leq id(v)$, the claim holds for the new message for analogous reasons. Therefore, the fourth statement holds in every state of $SUFFIX(S)$.

All in all, there is a state $S_2 \geq S_1$ as defined in the lemma. □

We continue with a basic observation that will turn out to be helpful for our proof of Lemma 5.6. This observation immediately follows from the pseudocode, which is why we state it without further justification.

**Observation 5.9.** *A leaving node does not remove a left neighbor from its neighborhood unless it executes* exit.

The following lemma will also turn out to be very helpful in various proofs throughout this section:

**Lemma 5.10.** *There is a state $S_3 \geq S_2$ such that in every state $S \geq S_3$ if there is a node $u$ such that* REVERSEANDLINEARIZE($nodeList, uniqueValue$) $\in u.Ch$ *and there is a node $v$ whose reference is contained in $Left(u) \cup Right(u) \cup u.Temp_L \cup u.Temp_R$ and $u.uniqueValues[v] = uniqueValue$, then $v$ is leaving.*

*Proof.* Let $S_3$ be the first state such that $S_3 \geq S_2$ and all REVERSEANDLINEARIZEACK() messages still in the system in $S_1$, all REVERSEANDLINEARIZE()

messages sent upon receipt of a ReverseAndLinearizeACK() message still in the system in $S_1$ and all ReverseAndLinearize() messages in the system in $S_1$ have been received. Assume for contradiction that there is a node $u$ in some state $S \geq S_3$ such that there is a ReverseAndLinearize($nodeList, uniqueValue$) message in $u.Ch$ and that there is a node $v$ whose reference is contained in $Left(u) \cup Right(u) \cup u.Temp_L \cup u.tempright$ and $u.uniqueValues[v] = uniqueValue$, but $v$ is staying. First of all, note that a ReverseAndLinearize($nodeList, uniqueValue$) message is sent only by a leaving node. Therefore, the message in $u.Ch$ must have been sent by a node $w \neq v$. Second, note that a ReverseAndLinearize($nodeList$, $uniqueValue$) message is sent to a node $u$ only upon receipt of a ReverseAndLinearizeACK($u, uniqueValue$) message. Note that according to the pseudocode, whenever a ReverseAndLinearizeACK($u, uniqueValue$) message is sent, the sender is $u$ itself. Thus, $u$ must have sent a ReverseAndLinearizeACK($v$, $uniqueValue$) message to $w$. Additionally note, however, that when this happens, $u$ ensures that $uniqueValue = u.uniqueValues[w]$. Since $uniqueValues$ are supposed to be unique after $S_1$ (because each node has executed Timeout at least once and new entries to $uniqueValue$ are chosen such that they are unique), $uniqueValue = u.uniqueValues[w]$ contradicts to $u.uniqueValues[v] = uniqueValue$. □

The next lemma states that if a node has a staying left neighbor in an infinite number of states then from some point on it will have one particular staying left neighbor in every state:

**Lemma 5.11.** *Let $v$ be an arbitrary staying node such that there exists a staying node $u$ such that $u \in Left(v)$ in an infinite number of states. Let $w$ be the node with the maximum identifier among all these nodes $u$. Then there is a state $S$ such that $w \in Left(v)$ in every state $S' \geq S$.*

*Proof.* Let $v$ and $w$ be defined as in the lemma and let $S$ be a state such that $S \geq S_3$ and that in every state $S' \geq S$ $v$ does not have any staying node $w'$ in $Left(v)$ with $id(w') > id(w)$. By the definition of $w$, such a state exists. Note that according to the pseudocode, $v$ does not send an Introduce($w, v$) message to a staying node after $S$. Further note that only staying nodes that receive an Introduce($w, v$) message send a Linearize($w$) message to $v$ (i.e., such a message is not sent at any other occasion). Consider the state $S' \geq S$ such that every Introduce() message still in some message channel in $S$ has been received and every Linearize() message sent as a response to an Introduce() message still in some message channel in $S$ has been received as well. By the aforementioned, there is no Linearize($w$) message in $v.Ch$ in any state after $S'$. Let $S'' \geq S'$ be the first state after $S'$ such that $w \in Left(v)$ again. Note that according to the pseudocode, there are exactly two cases in which $v$ removes a neighbor $w$: One is that $v$ received a ReverseAndLinearize($nodeList, uniqueValue$) message such that $v.uniqueValues[w] = uniqueValue$. According to Lemma 5.10, $w$ would be leaving in this case, which means this case does not apply. The other is

that $v$ received a LINEARIZE($w$) message. This cannot happen after $S'$. Thus, $w \in Left(v)$ in every state $S''' \geq S''$ and the proof is finished. □

Note that Lemma 5.11 immediately implies the following corollary:

**Corollary 5.12.** *There is a state $S_4 \geq S_3$ such that for every staying node $v$, either $v$ does not have a staying neighbor in $Left(v)$ in any state of $SUFFIX(S_4)$, or $v$ has a staying neighbor $w$ such that $w \in Left(v)$ in every state of $SUFFIX(S_4)$ and there is no node $w'$ such that $id(w') > id(w)$ and $w' \in Left(v)$ in any state of $SUFFIX(S_4)$.*

A key ingredient for the proof of Lemma 5.6 will be the definition of a node $u^*$: Let $u^*$ be the minimum node such that in $SUFFIX(S_2)$, (i) $u^*$ is leaving, and (ii) $u^*$ has no right incoming leaving neighbor. Note that such a node always exists by the definition of $S_1$ and that it is well-defined (and a fixed node) according to Lemma 5.8.

The next lemma we prove states that $u^*$ eventually will not have a right incoming neighbor at all (note that the definition of $u^*$ only requires $u$ to not have a right incoming *leaving* neighbor). It is proven by a non-trivial potential function argument involving two potential functions.

**Lemma 5.13.** *There is a state $S_5 \geq S_4$ such that $u^*$ does not have a right incoming neighbor.*

*Proof.* Since the potential function argument we use to prove the lemma is more involved than typical approaches, we need to introduce some definitions first. Let $\max_{intro}$ be the node $v$ with the greatest identifier such that $id(v) > id(u^*)$ and there is an $x$ such that INTRODUCE($u^*, v$) $\in x.Ch$. If there is no such node, $\max_{intro} := u^*$. Further, let $\max_{msgs}$ be the node $v$ with the greatest identifier such that $id(v) > id(u^*)$ and $(v, u^*) \in NG$. If no such node exists, let $\max_{msgs} := u^*$. Then $\max_{both}$ is defined as the node with the higher identifier among $\max_{intro}$ and $\max_{msgs}$. In addition to this, we define $count_{intro}$ as the number of INTRODUCE($u^*, v$) messages in the system such that $v = \max_{both}$ and $count_{msgs}$ as the number of edges $(v, u^*) \in NG$ such that $v := \max_{both}$. The first potential function we define is $\Phi_{id} := id(\max_{both})$. The second potential function $\Phi_{msg}$ is 0 if $\max_{both} = u^*$, or $\Phi_{msg} := count_{intro} + count_{msgs}$ otherwise.

We make use of these potential functions in the following way: First of all, we prove that $\Phi_{id}$ never increases and that $\Phi_{msg}$ increases only if $\Phi_{id}$ decreases. Second, we prove that as long as $\Phi_{id}$ is greater than $id(u^*)$ (note that it is lower bounded by that value), either of the two potential functions will decrease in finite time. These two statements together imply that eventually there will be no right incoming neighbor of $u^*$.

We begin with the former. Note that according to the pseudocode and to Lemma 5.8, the only case in which a staying node $v$ s.t. $id(v) > id(u^*)$ with an edge to $u^*$ sends $u^*$'s reference to a right neighbor $w$ after $S_2$ is when it receives an INTRODUCE($u^*, w$) message. In this case, to be precise, $v$ sends a LINEARIZE($u^*$)

message to $w$. By definition of $\Phi_{id}$ and $\Phi_{msg}$, both potentials do not increase in this case. In addition, note that whenever a staying node $w$ sends an INTRODUCE$(u^*, v)$ message, then $v = w$ and the recipient is closer to $u^*$ than $w$ itself. Since $w$ must have had an edge $(w, u^*)$ prior to that, if this increases $\Phi_{msg}$ then $\Phi_{id}$ decreases at the same time. Last, note that every leaving node $v$ such that $id(v) < id(u^*)$ never sends $u^*$'s reference to a right neighbor unless it leaves (which does not happen in $SUFFIX(S_1)$). Since no leaving node $v$ such that $id(v) > id(u^*)$ ever has an edge to $u^*$ by definition of $u^*$, we are finished proving that $\Phi_{id}$ never increases and that $\Phi_{msg}$ increases only if $\Phi_{id}$ decreases.

To prove that either of $\Phi_{id}$ and $\Phi_{msg}$ decreases in finite time unless $\Phi_{id} = id(u^*)$ already, first of all assume that there is a node $v$ such that $v = \max_{both}$ and that there is a node $x$ such that INTRODUCE$(u^*, v) \in x.Ch$. In this case, by definition of $\max_{both}$, there is no node $w$ such that $id(w) > id(v)$ and $(v, u^*) \in NG$. Upon receipt of the INTRODUCE$(u^*, v) \in x.Ch$ message, $x$ will send a LINEARIZE$(u^*)$ message to $v$. Note that this does not increase $\Phi_{msg}$. Upon receipt of this message, there are two cases. First, $v$ could have a left neighbor $w$ closer than $u^*$. In this case, $v$ will send a SINGLEINTRODUCE$(u^*)$ message to $w$ and not keep $u^*$'s reference, causing $\Phi_{msg}$ to decrease or (if $\Phi_{msg}$ would then be 0) causing $\Phi_{id}$ to decrease. Second, $v$ might not have such a left neighbor, in which case it adds $u^*$ to its neighborhood and we arrive at the second case.

Now suppose that there is a node $v$ such that $v = \max_{both}$ and $id(v) > u^*$ and $(v, u^*) \in NG$. Note that $v$ must be staying by the definition of $u^*$. We consider all possible types of messages containing $u^*$ and causing the edge $(v, u^*)$. Suppose $u^*$ is a first parameter of an INTRODUCE$(a, b)$ message in $v.Ch$. In that case, $id(b) \leq id(v)$ according to the definition of $\Phi_{msg}$. $v$ would then send $u^*$'s reference to $b$ and $\Phi_{id}$ or $\Phi_{msg}$ would directly decrease (if $id(b) < id(v)$ or that message would be turned into a LINEARIZE$(u^*)$ message that is put into $v.Ch$ (see below)). Now suppose $u^*$ is the second parameter of an INTRODUCE$(a, b)$ message in $v.Ch$. In that case, $v$ turns the message into a SINGLEINTRODUCE$(u^*)$ message (see below). If $u^*$ is the parameter of a LINEARIZE() message, either $(v, u^*)$ is turned into an explicit edge (see below), or SINGLEINTRODUCE$(u^*)$ is sent to a left neighbor of $v$ and the edge is not kept, in which case $\Phi_{id}$ or $\Phi_{msg}$ decrease. If $u^*$ is contained in a parameter of a REVERSEANDLINEARIZEACK() or a REVERSEANDLINEARIZE() message, either this message is turned into a SINGLEINTRODUCE() message (see below) or $(v, u^*)$ is turned into an explicit edge (see below). Note that according to Lemma 5.8, $u^*$ cannot be contained in a PROBESUCCESS() message and it cannot be the first parameter of a FORWARDPROBE() message. If $u^*$ is contained in the set $Next$ of a FORWARDPROBE$(source, destID, Next, seq)$ message, then according to Lemma 5.8, due to $u^* < id(v)$, $destID \leq id(v)$ must hold. According to the pseudocode, $v$ will then either send SINGLEINTRODUCE$(v)$ to itself (see below) or send a FORWARDPROBE$(source, destID, Next', seq)$ to a node in $Next' = Next \setminus \{v\} \cup Left(v)$, i.e., to a left neighbor of $v$. If $u^*$ is the parameter of a SINGLEINTRODUCE() message, either $(v, u^*)$ is turned into an explicit edge (see below) or SINGLEINTRODUCE$(u^*)$ is sent to a left neighbor of $v$ and the edge

is not kept, in which case $\Phi_{id}$ or $\Phi_{msg}$ decrease as well. Last, we consider the case that $(v, u^*)$ is an explicit edge.

If $(v, u^*)$ is an explicit edge, then during the next execution of TIMEOUT there are two cases: First, assume $v$ then has a left neighbor closer to $v$ than $u^*$. If the neighbor $w$ with the smallest identifier such that $id(u^*) < id(w)$ is leaving, $v$ would send $u^*$'s reference to $w$ contradicting the definition of $u^*$. If $w$ is staying, according to Corollary 5.12, $v$ will always have a staying left neighbor $w'$ such that $id(w') \geq id(w)$. During this execution of TIMEOUT, $v$ sends an INTRODUCE$(u^*, v)$ message to $w$, which $w$ will respond to with a LINEARIZE$(u^*)$ message to $v$. Since $v$ then still has a left neighbor $w''$ closer to $v$ than $u^*$ by the above, $v$ will then remove $u^*$ from $Left(v)$ and send a SINGLEINTRODUCE$(u^*)$ message to $w''$, yielding that $\Phi_{id}$ or $\Phi_{msg}$ decrease. Second, assume $v$ does not have a left neighbor closer to $v$ than $u^*$ during the next execution of TIMEOUT. In this case, $v$ will send a SINGLEINTRODUCE$(self)$ message to $u^*$, upon whose receipt $u^*$ will add $v$ to its right neighborhood. Note that $u^*$ does not remove a right non-leaving neighbor. Thus, during $u^*$'s next execution of TIMEOUT, $u^*$ will send a REVERSEANDLINEARIZEREQ$(left)$ message to $v$. Either $u^*$ has been removed from $Left(v)$ in the meantime, in which case $\Phi_{id}$ or $\Phi_{msg}$ have decreased and we are done, or $v$ will respond to $u^*$ with a REVERSEANDLINEARIZEACK$(v, uniqueValue)$ message such that $v.uniqueValues[u^*] = uniqueValue$. Upon receipt of that message, $u^*$ will respond with a REVERSEANDLINEARIZE$(NodeList, uniqueValue)$ message. When $v$ receives this message, either $u^*$ is no longer a neighbor of $v$, in which case $\Phi_{id}$ or $\Phi_{msg}$ have decreased, or $v$ will reverse its edge to $u^*$, in which case $\Phi_{id}$ or $\Phi_{msg}$ will also decrease.

All in all, $\Phi_{id}$ or $\Phi_{msg}$ decreases in finite time. By the aforementioned, this finishes the proof that $u^*$ will not have a right incoming neighbor from some state $S_5$ onwards. □

In the remainder of the proof of Lemma 5.6 we prove that $u^*$ also does not have any left incoming neighbor. As an intermediate step, we prove the following lemma:

**Lemma 5.14.** *Let $L$ be the set of all leaving nodes $v$ such that $id(v) \leq id(u^*)$. There is a state $S_L \geq S_3$ such that after $S_L$ whenever a node $u$ sends the reference of a node $v \in L$ to some node $w$ such that $id(w) < id(v)$, then $id(u) < id(w)$ or* INTRODUCE$(v, w) \in u.Ch$ *in the state before.*

*Proof.* At first, we prove that there is a state $S$ such that for every leaving node $v'$ if $v'$ has a right neighbor $v \in L$, then $v \notin Right(v')$ in every state $S' \geq S$ (i.e., $(v', v)$ is an implicit edge or $v \in v'.Temp_R$). Note that a leaving node $v'$ never adds a node to $Right(v')$. Therefore, it is sufficient to show that every $v \in L$ will eventually be removed from $Right(v')$ for every leaving node $v'$. Consider an arbitrary pair of nodes $v'$ and $v$ such that in some state $S' \geq S_3$, $v'$ is leaving, $v \in L$ and $v \in Right(v')$. Note that then $id(v') < id(v)$ according to Lemma 5.3, implying $id(v') < id(u^*)$. Suppose for contradiction that $v$ is not removed from

$Right(v')$ in any later state. According to Lemma 5.8, the fact that $id(v') < id(u^*)$ and due to the definition of $u^*$, $v'$ has a leaving right incoming neighbor in every state after $S'$. At some point in time, during the execution of TIMEOUT, that node will send a REVERSEANDLINEARIZEREQ($right$) message to $v'$ causing $v'$ to send a REVERSEANDLINEARIZEACK($v', uniqueValue$) message to $v$ such that $uniqueValue = v'.uniqueValue[v]$. Upon receipt, $v$ will respond with a REVERSEANDLINEARIZE($nodeList, uniqueValue$) message causing $v'$ to remove $v$ from $Right(v)$, which represents a contradiction. Thus, there is a state $S$ as specified above.

Next consider the state $S_L \geq S$ such that all REVERSEANDLINEARIZE() messages still in the incoming channel of any node in $S$ have been received. Note that the first parameter of every REVERSEANDLINEARIZE() message only contains a node from the set $Right$ of the sending node. Thus, in every state $S' \geq S_L$ there will be no REVERSEANDLINEARIZE($nodeList, uniqueValue$) message such that $v \in nodeList$ for any $v \in L$. We now argue that after $S_L$, every node $u$ only sends a reference of a node $v \in L$ to a node $w$ such that $id(w) < id(v)$ if $u$ received an INTRODUCE($v, w$) message, or $id(u) < id(w)$.

Observe in the pseudocode in Listing 5.1 that in the TIMEOUT, LINEARIZE(), and SINGLEINTRODUCE() actions, unless a node exits (which cannot happen after $S_1$), each node $u$ that sends a reference $v$ to a left neighbor $w$ does this only if $id(v) < id(w) < id(u)$. Furthermore, in the INTRODUCE() action, a node sends a reference only to another node if that node is the second parameter and in this case it sends the first parameter to that node. So if $u$ sends $v$'s reference to $w$, it would have received an INTRODUCE($v, w$) message before. Next, note that in the REVERSEANDLINEARIZEREQ() action, a node $v$ may possibly send its own reference to another node $w$ but if $v$ is leaving then only to the right, i.e., when $id(v) < id(w)$. If some node $u$ executes the REVERSEANDLINEARIZEACK() action, then if $u$ is leaving it might send a set of references of nodes with a greater identifier than $u$ to the first parameter of the REVERSEANDLINEARIZEACK() message. Note that by definition of $S_L$ this set may not contain any $v \in L$. Moreover, in the REVERSEANDLINEARIZE($nodeList, uniqueValue$) action, a leaving node $v$ sends its own reference to another node $w$ only if $id(w) > id(v)$. Last, note that by Lemma 5.8, $v$ cannot be contained in a PROBESUCCESS() message or as the first parameter of a FORWARDPROBE() message. Assume $v$ is contained in the set $Next$ of a FORWARDPROBE($source, destID, Next, seq$) message in $u.Ch$ for some node $u$. If $u$ does not send a new FORWARDPROBE() message, it sends a FORWARDPROBE($v$) message to itself, which leads to a case we handled before. Otherwise, according to the pseudocode, $u$ might send a new FORWARDPROBE($source, destID, Next', seq$) message, in which $Next'$ might contain $v$. Note that if $destID > id(u)$, by Lemma 5.8 $Next$ would contain only references of nodes $w'$ such that $id(w') > id(u)$ and, according to the pseudocode, $Next'$ would only contain nodes from $Next$ and $Right(w)$ and the new FORWARDPROBE() message would then be sent to a node $w \in Next'$ such that $id(w) > id(u)$. Thus, assume $destID < id(u)$. In this case, by Lemma 5.8, for all nodes $w' \in Next$, $id(w') < id(u)$. Note

that $Next' := Next \setminus \{u\} \cup \{w' \in Left(u) : id(w') \geq destID\}$ and the new FORWARDPROBE() message would be sent to the node $w$ with the maximum identifier among all nodes in $Next'$. Thus, since $v \in Next$, then $id(v) \leq id(w)$. All in all, the claim of the lemma is proven. □

The following lemma states that each staying node $u$ always keeps the closest right neighbor it ever has (and that it will eventually have an explicit edge to this node):

**Lemma 5.15.** *For an arbitrary staying node $u$ and an arbitrary state $S^* \geq S_3$ let $w$ be the node with the minimum identifier among all nodes $w'$ such that (i) $id(w') > id(u)$, and (ii) there is an edge $(u, w')$ in some state $S \geq S^*$. If $w$ is staying as well, there is a state $S' \geq S^*$ such that $(u, w)$ will be an explicit edge in every state $S'' \geq S'$.*

*Proof.* Let $u$ be an arbitrary staying node and let $w$ be defined as in the claim of the lemma and let $S \geq S^*$ be a state such that $(u, w)$ exists in $S$. Assume that $w$ is staying. First of all, note that as soon as $(u, w)$ is an explicit edge, it is never removed: According to the pseudocode, a staying node $u$ removes a node from $Right$ only if it has a right neighbor closer to itself (see the LINEARIZE() action), which cannot be the case for $w$ by definition of $w$, or if it receives a REVERSEANDLINEARIZE($nodeList, uniqueValue$) message such that $u.uniqueValues[w] = uniqueValue$, which by Lemma 5.10 cannot be for a staying node $w$.

Second, note that whenever a staying node $u$ receives a reference of another node $w$ which is closer than its current closest right neighbor, it either adds this reference to $Right(u)$ immediately or sends a SINGLEINTRODUCE($w$) message to itself upon whose receipt it will add $w$'s reference to $Right(u)$. Thus, since $(u, w)$ exists in $S$, there is a state $S' \geq S$ such that $(u, w)$ will be an explicit edge. By the aforementioned, $(u, w)$ will be an explicit edge in every state $S'' \geq S'$. □

The previous two lemmas are actually auxiliary lemmas to prove the next two lemmas, which together imply that $u^*$ will not have a left incoming neighbor at some point. Before we prove them, we introduce the following notion:

**Definition 5.16** (Relevant node)**.** *A node $x$ such that $id(x) < id(u^*)$ is called* relevant *if and only if there is a leaving node $v$ with $id(x) < id(v) \leq id(u^*)$ and (i) there is an edge $(x, v)$ in $NG$, or (ii) there is a node $y$ such that* INTRODUCE($v, x$) $\in y.Ch$.

The next lemma tells us that a node with an identifier smaller than that of the "leftmost" relevant node cannot become relevant.

**Lemma 5.17.** *For some arbitrary state $S \geq S_L$, if $S$ contains a relevant node let $x$ be the relevant node with the minimum identifier in $S$. Otherwise, let $x = u^*$. In every state $S' \geq S$ there is no node $z$ such that $id(z) < id(x)$ and $z$ is relevant.*

*Proof.* Let $S \geq S_L$ be arbitrary and let $x$ be defined as in the claim of the lemma. We begin with proving that no node $z$ with an identifier smaller than that of $x$ will become relevant in any state after $S$.

Therefore assume there is a node $z$ with $id(z) < id(x)$ that becomes relevant after $S$ and let $z$ be the first such node. Assume it becomes relevant during some state $S'$. We consider the cases how this could have happened. First, note that a ReverseAndLinearizeACK$(z, uniqueValue)$ message is only sent by $z$ itself. So for a ReverseAndLinearizeACK$(z, uniqueValue)$ message added to the incoming channel of a leaving node $v$, $z$ would need to have an edge to $v$, in which case (i) would have been true before. But then $z$ would not have become relevant during $S'$. Second, note that an Introduce$(v, z)$ message is only sent by $z$ itself, too. Thus, for the same reasons the occurrence of such a message cannot cause $z$ to become relevant. Third, assume that for some leaving node $v$ such that $id(v) \leq id(u^*)$, an edge $(z, v)$ is established that did not exist before. This means some node $u$ sent $v$'s reference to $z$. Note that $id(u) > id(z)$ by definition of $z$ must hold (otherwise, (i) would be contradicted). Furthermore, $u$ cannot have received an Introduce$(v, z)$ message (otherwise (ii) would have been contradicted). This, however, represents a contradiction to Lemma 5.14. Thus, all in all, $z$ cannot become relevant and the first claim is proven. □

As the last in a sequence of lemmas to show that $u^*$ will not have an incoming neighbor at some point in time, we prove that every relevant node will eventually stop being relevant. In other words, together with the previous lemma, the next lemma shows that the function denoting the identifier of the "leftmost" relevant node is monotonically increasing (as long as there is a relevant node).

**Lemma 5.18.** *For some arbitrary state $S \geq S_L$ such that a relevant node exists in $S$, let $x$ be the relevant node with the minimum identifier in $S$. Then there is a state $S' \geq S$ such that $x$ is not relevant in $S'$.*

*Proof.* Let $S \geq S_L$ be arbitrary such that a relevant node exists in $S$ and let $x$ be the relevant node with the minimum identifier in $S$. We show that $x$ will eventually stop being relevant.

Let $\phi_{rl}$ be a function that returns the leaving node $v$ with the minimum identifier such that $id(x) < id(v) \leq id(u^*)$ and there is an edge $(x, v)$ or an Introduce$(v, x) \in y.Ch$ for some node $y$ in $S$. Note that according to Lemma 5.14 and the definition of $x$ and Lemma 5.17, no node can introduce a leaving node $v'$ such that $id(v') < \phi_{rl}$ to $x$. Therefore, $\phi_{rl}$ is monotonically increasing. We will show that as long as $x$ remains relevant, $\phi_{rl}$ will eventually increase, which is sufficient as $\phi_{rl}$ is naturally bounded from above. In the following, let $v = \phi_{rl}$. During the proof, we will show that there is a state $S'$ such that there is no Introduce$(v, x) \in y.Ch$ for any node $y$ in any state $S'' > S'$ (*). Note that after $S'$, no node can send $v$'s reference to $x$ again: i.e., as soon as this is the case, the number of implicit edges $(x, v)$ is monotonically decreasing. Note that since each message will eventually be received, that number is also strictly monotonically decreasing: i.e., there will

be a state $S''' > S''$ such that no implicit edges $(x, v)$ will exist in any state after $S'''$. If the last explicit edge $(x, v)$ is removed at some point after $S'''$, we are done proving that $\phi_{rl}$ will increase. We continue by showing that if this is not the case, $x$ will receive a REVERSEANDLINEARIZEREQ($right$) message at some point in time (**). This will cause $x$ to send a REVERSEANDLINEARIZEACK($x, uniqueValue$) message to $v$ with $uniqueValue = x.uniqueValues[v]$. Upon receipt of that message, $v$ will respond to $x$ with a REVERSEANDLINEARIZE($nodeList, uniqueValue$) message, which will cause $x$ to remove the edge $(x, v)$, and we are done proving that $\phi_{rl}$ will increase.

We distinguish three cases and will show that (*) and (**) (if necessary) hold true.

First, assume $x$ is leaving. Note that a leaving node never sends an INTRODUCE() message with itself as any parameter to any other node. Thus, as soon as all INTRODUCE($v, x$) messages existing in any incoming channels in $S$ have been received, there will be no more such messages and (*) is proven. Additionally, since $id(x) < id(u^*)$ by definition, $x$ has a right incoming neighbor throughout $SUFFIX(S_3)$, which will at some point in time after $S'''$ send a REVERSEANDLINEARIZEREQ($right$) message to $x$, proving (**). We are thus done proving that $\phi_{rl}$ will increase in this case.

Second, assume $x$ is staying and $v$ is the node with the minimum identifier that is a right outgoing neighbor of $x$ in any state $S' \geq S$. Note that according to the pseudocode, $x$ will not send an INTRODUCE($v, x$) message to any node after $S$ and therefore no such message will ever be created, proving (*). Now assume the explicit edge $(v, x)$ will never be removed after $S'''$. According to the pseudocode, $x$ will at some point send SINGLEINTRODUCE($self$) to $v$ after which we know that $v$ stores $x$'s reference in $v.Temp_L$. By Lemma 5.8, $x$ will always remain a left neighbor of $v$ and therefore, during some execution of TIMEOUT, $v$ will send a REVERSEANDLINEARIZEREQ($right$) message to $x$, proving (**). We are thus done proving that $\phi_{rl}$ will increase in this case as well.

Third, assume $x$ is staying and that there is a node $w'$ such that $id(x) < id(w') < id(v)$ and that there is a state after $S$ in which there is an edge $(x, w')$. Let $w$ be the node with the minimum identifier fulfilling this property. Note that as argued before, $w$ must be staying, since otherwise $x$ would have a leaving right neighbor with an identifier smaller than $\phi_{rl}$, representing a contradiction. Therefore, we can apply Lemma 5.15 to obtain that the explicit edge $(x, w)$ exists forever from some state $S' \geq S$. Note that after $S'$, according to the pseudocode if $x$ receives $v$'s reference, it will not add $v$ to $Right(x)$ anymore. Thus, we show that $v \notin Right(x)$ at some point after $S'$, which then holds in every subsequent state. Since the explicit edge $(x, w)$ exists throughout $SUFFIX(S')$, we know that during TIMEOUT, $x$ will send an INTRODUCE($v, x$) to some right neighbor $w'$ such that $id(x) < id(w') < id(v)$. Note that $w'$ must be staying by the definition of $v$. Therefore, according to the pseudocode, $w'$ will respond with a LINEARIZE($v$) message. Upon receipt of that message, $x$ still has at least one right neighbor closer to $x$ than $v$ (because the explicit edge $(x, w)$ exists then) and will thus

remove $v$ from $Right(x)$ according to the pseudocode. As argued before, $v$ will never be added to $Right(x)$ anymore. Thus, $x$ will never send an INTRODUCE$(v, x)$ message anymore and as soon as all these messages still existing in the system have been received, (*) holds. As argued before, all implicit edges $(x, v)$ will eventually vanish and no such edge can come up anymore. Since we have already proven that there will not be an explicit edge $(x, v)$ anymore, we are done proving that $\phi_{rl}$ will eventually increase.

As argued before, $x$ will eventually stop being relevant, which completes the proof of this lemma. □

Note that Lemma 5.17 and Lemma 5.18 imply the following corollary:

**Corollary 5.19.** *There is a state $S$ such that $u^*$ does not have a left incoming neighbor in $SUFFIX(S)$.*

This enables us to finally prove Lemma 5.6, which we restate as follows:

**Lemma 5.6.** *For any computation of* BUILD-LIST* *there exists a computation suffix in which all leaving nodes are inactive.*

*Proof.* Assume for contradiction that there is a leaving node in $C$ that will not exit at any point in time. Then there exists a node $u^*$ as defined before. Lemma 5.13 and Corollary 5.19 together imply that there is a state $S_6$ such that $u^*$ will not have an incoming neighbor at all. After all messages still in $u^*.Ch$ during $S_6$ have been received, $\mathcal{NIDEC}$ will be true for $u^*$. This means that upon the next execution of TIMEOUT, $u^*$ would execute **exit**. However, $u^*$ was defined as a node that had not left before $S_1$ and $S_1$ was defined such that no node executes **exit** in $SUFFIX(S_1)$. Therefore, we have a contradiction to the assumption that there is a leaving node in $C$ that will not exit at any point in time. □

Lemma 5.5, Lemma 5.6 and the fact that no staying node ever executes exit in BUILD-LIST* yield Theorem 5.4, which finishes this section.

## 5.4. Build-List* Self-Stabilizes to the Line Topology

In this section we prove that BUILD-LIST* self-stabilizes to the line topology. Therefore, the main theorem of this section is as follows:

**Theorem 5.20.** BUILD-LIST* *self-stabilizes to the line topology.*

Note that although this theorem also appears in [SSS15], its proof has been rewritten entirely. Whereas that proof is mainly built upon a result contained in the first part of [SSS15], which is not contained in this chapter, our proof follows a new structure and makes use of some lemmas proven in Section 5.3.

The main idea of the proof is as follows: First of all, we consider a fixed computation again and start our consideration at the point at which all leaving

nodes have left (which will happen according to Theorem 5.4). With the help of Corollary 5.12 and Lemma 5.15 from Section 5.3, we show that eventually each node has at most one neighbor in either direction in $ENG$ and that $ENG$ will remain unchanged from some point. We then continue by proving several properties of $ENG$ in such a state: First, we show that $ENG$ is bidirected (meaning that whenever there is an edge $(u, v)$ in $ENG$ then there is also an edge $(v, u)$). After that, we show that $ENG$ is strongly connected (meaning that there is a path between any pair of nodes in $ENG$). Using these two results, we can establish a simple contradiction to the assumption that $ENG$ does not form a line: We consider two nodes that are neighbors in the line but not in $ENG$ and consider a shortest path between them (which exists since $ENG$ is strongly connected). This path cannot go straight into one direction, i.e., only to the right or only to the left. Instead, at some node it must change its direction. Such a node, however, would need to have two neighbors in $ENG$ on one side, which represents a contradiction.

For the rest of this section, we consider an arbitrary but fixed computation $C$ of BUILD-LIST* that starts in a state in which $PNG$ is weakly connected. The following is a corollary of Lemma 5.5 and Lemma 5.6:

**Corollary 5.21.** *There is a state $S_1$ in which all leaving nodes are inactive and $PNG$ is weakly connected throughout $SUFFIX(S_1)$.*

This insight allows us to consider this suffix only and thus to ignore the leaving nodes. We continue with the following lemma:

**Lemma 5.22.** *There is a state $S_2 \geq S_1$ such that throughout $SUFFIX(S_2)$, $ENG$ does not change and each node $v$ has at most a single fixed node in $Left(v)$ and at most a single fixed node in $Right(v)$.*

*Proof.* From Corollary 5.12, we obtain that there is a state $S_l \geq S_1$ such that in $SUFFIX(S_l)$ every node $v$ either has no left neighbor in $Left(v)$ at all or has a left neighbor $u$ in $Left(v)$ throughout $SUFFIX(S_l)$ such that it will never have a closer neighbor in $Left(v)$ throughout $SUFFIX(S_l)$. Note that according to the pseudocode after $S_l$ no node will add any node to $Left$ during the execution of LINEARIZE(), SINGLEINTRODUCE(), or FORWARDPROBE() anymore. Thus, the only occasion where a node is added to $Left$ again, is the INTRODUCE() action. For each node $u$, let $mri(u) := \max\{w \in V : id(w) > id(u) \wedge (\exists\, u' \in V : (w, u') \in ENG \wedge id(u) < id(u') < id(w)) \wedge ((w, u) \in ENG) \vee (\exists\, x \in V : \text{INTRODUCE}(u, x) \in w.Ch))\}$. Note that whenever INTRODUCE$(u, x)$ messages are sent to a node $w$ such that $id(u) < id(w)$ then $x$ is the sender, $id(x) > id(w)$, and there was an explicit edge $(x, u)$ right before and $x$ had at least one neighbor $u'$ such that $id(u) < id(u') < id(x)$ (one such it will always have by Corollary 5.12). Therefore, $id(mri(u))$ cannot increase and as soon as $mri(u)$ removes $u$ from $Left$, it will not add $u$ to $Left$ again. Hence, we argue that $mri(u)$ removes $u$ from $Left$ at some point: Note that during TIMEOUT, $mri(u)$ will send INTRODUCE$(u, mri(u))$ to some left neighbor $u'$. This node will respond with a LINEARIZE$(u)$ message to

$mri(u)$. Upon receipt of that message, $mri(u)$ will remove $u$ from $Left$, so $mri(u)$ will change and $id(mri(u))$ will decrease. Note that this proves that every node $v$ will in some state $S'_l$ have either no or a fixed single left neighbor in $Left(v)$ throughout $SUFFIX(S'_l)$.

Similar to the above, from Lemma 5.15 we obtain that there is also a state $S_r \geq S_1$ such that in $SUFFIX(S_r)$ every node $v$ either has no left neighbor at all or has a right neighbor $w$ in $Right(v)$ such that it will never have a closer neighbor throughout $SUFFIX(S_r)$ and $w \in Right(v)$ throughout $SUFFIX(S_r)$. Since the protocol is completely symmetric for staying nodes, for similar arguments as in the last case, we obtain that every node $v$ will in some state $S'_r$ have either no or a fixed single right neighbor in $Right(v)$ throughout $SUFFIX(S'_r)$.

Therefore, let $S_2$ be an arbitrary state after both $S'_l$ and $S'_r$ and the lemma is proven. □

We continue by showing some properties of $ENG$ in $SUFFIX(S_2)$. The first is given by the following lemma: We begin with proving that in every state of $SUFFIX(S_2)$, the graph of explicit edges is bidirected, i.e., for every explicit edge $(u, v) \in ENG$, also $(v, u) \in ENG$:

**Lemma 5.23.** *$ENG$ is bidirected in $SUFFIX(S_2)$.*

*Proof.* Assume for contradiction that the claim does not hold: i.e., there is a pair of nodes in a state of $SUFFIX(S_2)$ such that $v \in Right(u)$ and $u \notin Left(v)$, or $u \in Left(v)$ and $v \notin Right(u)$. Suppose the former (the other case is analogous). First of all, note that during Timeout, $u$ will send a SingleIntroduce($u$) message to $v$. We know that there must be a node $w \in Left(v)$ such that $id(w) > id(u)$, because otherwise $v$ would add $u$ to $Left(v)$ yielding a contradiction to Lemma 5.22. Thus, upon receipt of that message, $v$ will send a SingleIntroduce($u$) message to $w$. Observe in the pseudocode that each intermediate node $w'$ that does not store $u$ will send a SingleIntroduce($u$) message to a node whose identifier is between $w'$ and $u$. This implies that eventually a node $w''$ such that $id(u) < id(w'') < id(v)$ will receive a SingleIntroduce($u$) message and not send a new SingleIntroduce($u$) message: i.e., it will keep $u$'s reference in $Left(w'')$ (since $ENG$ does not change according to Lemma 5.22, there must have been such an edge before, by the way). During the next execution of Timeout, $w''$ will send a SingleIntroduce($w''$) message to $u$. Since $id(u) < id(w'') < id(v)$, $u$ would add $w''$ to $Right(u)$ yielding the desired contradiction to Lemma 5.22. Thus, $ENG$ is bidirected in $SUFFIX(S_2)$. □

Before we can show that $ENG$ is also connected, we prove the following auxiliary lemma, which is straightforward given Lemma 5.22:

**Lemma 5.24.** *There is a state $S_3 \geq S_2$ such that in $SUFFIX(S_3)$ there does not exist any* Introduce($v, w$) *or* Linearize($v$) *message.*

*Proof.* Notice that INTRODUCE$(v, w)$ messages are not sent when each node has at most one left and at most one right explicit neighbor (which is the case in $SUFFIX(S_2)$ according to Lemma 5.22). Besides, LINEARIZE$(v)$ messages are only sent during the INTRODUCE$(v, w)$ action. Thus, there is a state $S_3 \geq S_2$ such that in $SUFFIX(S_3)$, no INTRODUCE$(v, w)$ and LINEARIZE$(v)$ messages will exist. □

The previous lemmas enable us to prove that the graph of explicit edges is strongly connected in every state of $SUFFIX(S_2)$:

**Lemma 5.25.** *$ENG$ is strongly connected in $SUFFIX(S_2)$.*

*Proof.* The idea of this proof is to show that there is a state $S_4$ after $S_3$ such that whenever there is an implicit edge $(u, v)$ in $SUFFIX(S_4)$ then there is also an undirected path via explicit edges from $u$ to $v$.[1] Since $NG$ is weakly connected throughout $SUFFIX(S_1)$ according to Corollary 5.21 and because $ENG$ is bidirected in $SUFFIX(S_2)$ according to Lemma 5.23, then $ENG$ must be strongly connected in $SUFFIX(S_4)$. However, since $ENG$ remains unchanged throughout $SUFFIX(S_2)$ (according to Lemma 5.22), this must hold for $ENG$ in every state of $SUFFIX(S_2)$ and the claim follows.

Note that newly initiated FORWARDPROBE() messages are initiated with the third parameter $Next = \{self\}$. Since according to Lemma 5.22 in $SUFFIX(S_2)$ there is at most one node in $Left(v)$ and in $Right(v)$ for every node $v$ and since $Left(v)$ and $Right(v)$ do not change anymore, and since the FORWARDPROBE() action removes the current node from $Next$ and adds nodes from either $Left$ or $Right$ to $Next$ and then sends it to a node in $Next$, every FORWARDPROBE$(source, destID, Next, seq)$ message in the channel of a node $x$ caused by a FORWARDPROBE() message initiated after $S_2$ has the property that $Next = \{x\}$ and there is an explicit edge from $source$ to $x$. Consider $S' \geq S_3$ as a state in which all FORWARDPROBE() messages initiated before $S_2$ are inactive. In every state of $SUFFIX(S')$ for all implicit edges $(u, v)$ that are caused by FORWARDPROBE() messages, there is an undirected path via explicit edges from $u$ to $v$. Furthermore, for every FORWARDPROBE$(source, destID, Next, seq)$ message in $x.Ch$ for some node $x$ in $SUFFIX(S')$, $Next = \{x\}$. Since a PROBESUCCESS$(destID, dest)$ message is sent to a node $source$ only upon receipt of a FORWARDPROBE$(source, destID, Next, seq)$ message by the only node $x \in Next$ and such that $dest = x$, we similarly obtain that there is a state $S'' \geq S'$ such that in $SUFFIX(S'')$ for every implicit edge $(u, v)$ caused by PROBESUCCESS() message, there is an undirected path via explicit edges from $u$ to $v$. Last, note that SINGLEINTRODUCE$(u)$ messages are sent only in the following cases: (i) during TIMEOUT, in which case there must be an explicit edge from the sending node $u$ to the receiving node $v$, (ii) during INTRODUCE() or LINEARIZE(), which

[1] In a directed graph $G$, an *undirected path* from some node $u$ to some node $v$ is a sequence of edges $(u = w_1, w_2, w_3, \ldots, w_k = v)$ such that for every two consecutive nodes $w_i$ and $w_{i+1}$ at least one of the edges $(w_i, w_{i+1})$ or $(w_{i+1}, w_i)$ exist in $G$.

does not occur in $SUFFIX(S_3)$ according to Lemma 5.24, (iii) during ForwardProbe() or ProbeSuccess() to the sending node $v$ itself, in which case we know that there is an undirected path via explicit edges from $u$ to $v$, or (iv) during SingleIntroduce($u$), which is the case that requires some elaboration. Note that in case (iv) there is an explicit edge from the sending node $v$ to the receiving node $w$. Furthermore, note that the sender $v$ of a SingleIntroduce($u$) sends the message to a node $w$ whose identifier is closer to $v$ than $u$. This inductively implies that every SingleIntroduce($u$) message in the channel of any node in $S''$ will cause only a finite number of other messages, so there is a state $S_4 \geq S''$ such that in $SUFFIX(S_4)$ for every implicit edge $(u, v)$ caused by a SingleIntroduce() message, there is an undirected path via explicit edges from $u$ to $v$. Note that all in all, in $SUFFIX(S_4)$ for every implicit edge $(u, v)$ there is also an undirected path via explicit edges from $u$ to $v$. As argued at the beginning of this proof, this implies that $ENG$ is strongly connected in every state of $SUFFIX(S_2)$. □

After having proven the previous lemmas, we are ready to prove Theorem 5.20, which we restate as follows:

**Theorem 5.20.** Build-List* *self-stabilizes to the line topology.*

*Proof.* Assume for contradiction that in some state of $SUFFIX(S_2)$ there is a pair of nodes $u$ and $v$ such that $u$ and $v$ are neighbors in the line topology: i.e., there is no node $w$ such that $id(u) < id(w) < id(v)$ (w.l.o.g. assume $id(u) < id(v)$), but $u$ and $v$ are not neighbors in $ENG$. Since $ENG$ is bidirected and strongly connected according to Lemma 5.23 and Lemma 5.25, there is a shortest path $P = (u = x_1, x_2, \ldots, x_k = v)$ from $u$ to $v$ via explicit edges such that also $(x_{i+1}, x_i) \in ENG$ for every $1 \leq i < k$. Note that since $k \geq 3$ there must be a node $x_i$ at which the path "changes its direction": i.e., $id(x_{i-1}) > id(x_i)$ and $id(x_{i+1}) > id(x_i)$, or $id(x_{i-1}) < id(x_i)$ and $id(x_{i+1}) < id(x_i)$. However, since $ENG$ is bidirected this would imply that $x_i$ has two neighbors in $Left$ or $Right$, representing a contradiction to Lemma 5.22. Thus, we obtain that in $SUFFIX(S_2)$, the nodes form a line, which finishes the proof of Theorem 5.20. □

## 5.5. Build-List* Satisfies Monotonic Searchability

As explained in Chapter 4, it is generally not possible for a protocol to unconditionally satisfy monotonic searchability. Therefore, we again define a set of invariants that need to hold (and that will hold in every computation eventually) for monotonic searchability to be possible. Later on we prove that Build-List* satisfies monotonic searchability according to these invariants. In Section 5.5.1, we introduce some additional definitions used in this section only and state the main theorem of this section. We then prove that the invariants will eventually hold forever in every computation in Section 5.5.2. After that, in Section 5.5.3, we prove that Build-List* satisfies monotonic searchability in the suffixes consisting of admissible states only.

### 5.5.1. Definitions and Main Results

Before we define the set of invariants used for BUILD-LIST*, we introduce some additional notation that will aid us throughout this section. We will refer to the notion of a straight path again, which was introduced in Definition 4.4 in Chapter 4. Note that the definition of a straight path uses the notion of a search edge. To apply it to this chapter, we define all explicit edges to be search edges.

**Definition 5.26** ($R_s(v)$, $R_s^+(ID, w)$, $R_s^+(ID, U)$)**.** *We define the following sets:*

$\mathbf{R_s(v)}$ *For an arbitrary node $v$, $R_s(v)$ consists of all staying nodes $x$ such that there is a (possibly empty) straight path from $v$ to $x$.*

$\mathbf{R_s^+(ID, w)}$ *For an arbitrary identifier $ID$ and an arbitrary node $w$, if $id(w) > ID$ then $R_s^+(ID, w) := \{u \in R_s(w) : id(u) \geq id(w)\}$, if $id(w) < ID$, then $R_s^+(ID, w) := \{u \in R_s(w) : id(u) \leq id(w)\}$ and if $id(w) = ID$, then $R_s^+(ID, w) := R_s(w)$.*

$\mathbf{R_s^+(ID, U)}$ *For an arbitrary identifier $ID$ and an arbitrary set $U$ of node references, $R_s^+(ID, U) := \bigcup_{u \in U} R_s^+(ID, u)$.*

Using these definitions, we can establish the message invariants used for the definition of an admissible state.

**Definition 5.27** (Invariants for BUILD-LIST*)**.** *We define the following invariants, in which we require that $v$, $w$, source and dest are node references, $uniqueValue$ and seq are numbers, $nodeList$ and $Next$ are sets of node references, and $destID$ is an identifier.*

1. *If there is an* INTRODUCE$(v, w)$ *message in $u.Ch$, then* $\min(id(v), id(w)) < id(u) < \max(id(v), id(w))$*, and $R_s^+(id(w), u) \subseteq R_s(w)$.*

2. *If there is a* LINEARIZE$(v)$ *message in $w.Ch$, then there is a node $u$ such that* $\min(id(v), id(w)) < id(u) < \max(id(v), id(w))$*, $u \in Right(w) \cup Left(w)$, and $R_s^+(id(w), v) \subseteq R_s(u)$.*

3. *If there is a* REVERSEANDLINEARIZEACK$(v, uniqueValue)$ *message in $u.Ch$, then $u \neq v$ and $v.uniqueValues[u] = uniqueValue$ and $u$ is the only node $u'$ such that $v.uniqueValues[u'] = uniqueValue$.*

4. *If there is a* REVERSEANDLINEARIZE$(nodeList, uniqueValue)$ *message in $u.Ch$, then there is exactly one node $v$ such that $u.uniqueValues[v] = uniqueValue$ and $v$ is leaving. Furthermore, if $id(v) < id(u)$ then $\forall\, x \in nodeList : id(x) < id(v)$ and if $id(v) > id(u)$ then $\forall\, x \in nodeList : id(x) > id(v)$. Moreover, $R_s^+(id(u), v) = R_s^+(id(u), nodeList)$.*

5. *If there is a* FORWARDPROBE$(source, destID, Next, seq)$ *message in $u.Ch$, then*

   a) $id(source) \leq id(u) \leq destID$ *and* $\forall\, x \in Next : id(x) \geq id(u)$ *and* $u = \text{argmin}_{u' \in Next}(id(u'))$, *or* $destID \leq id(u) \leq id(source)$ *and* $\forall\, x \in Next : id(x) \leq id(u)$ *and* $u = \text{argmax}_{u' \in Next}(id(u'))$,

   b) $R_s^+(id(source), Next) \subseteq R_s(source)$,

   c) *if there is a node* $v$ *such that* $id(v) = destID$, $v$ *is staying and* $v \notin R_s^+(id(source), Next)$, *then for every admissible state in which* $source.lseq[destID] < seq$, $v \notin R_s(source)$.

6. *If there is a* ProbeSuccess$(destID, dest)$ *message in* $u.Ch$, *then* $id(dest) = destID$ *and either* $dest \in R_s(u)$ *or* $dest$ *is leaving.*

7. *If there is a* ProbeFail$(destID, seq)$ *message in* $u.Ch$, *then either there is no staying node with identifier* $destID$, *or for every admissible state in which* $u.lseq[destID] < seq$ *and for the node* $v$ *with* $id(v) = destID$, $v \notin R_s(u)$.

8. *If there is a* Search$(v, destID)$ *message in* $u.Ch$ *and* $u$ *is staying, then* $id(u) = destID$ *and* $u \in R_s(v)$.

Formally, we define an admissible state as follows:

**Definition 5.28** (Admissible State)**.** *A state* $S$ *is called* admissible *if and only if all of Invariants 1-8 (c.f. Definition 5.27) hold.*

Armed with these definitions, we now formally define the main theorem of this section, whose proof makes up the rest of this chapter:

**Theorem 5.29.** Build-List* *admissible-message satisfies monotonic searchability according to* Search*.

### 5.5.2. Proving that Every Computation Has an Admissible Suffix

The main result of this subsection is formalized by the following lemma:

**Lemma 5.30.** *Every computation* $C$ *of* Build-List* *contains a state* $S$ *such that every state* $S' \geq S$ *is admissible.*

To prove that every computation of Build-List* contains a suffix in which every state is admissible, similar to Section 4.4.3 in Chapter 4 we first stepwise prove that the subsequent state of every admissible state is admissible as well. After that we show that every computation of Build-List* contains an admissible state.

We begin with the first four invariants for the step-by-step proof:

**Lemma 5.31.** *In every computation of* Build-List*, *if Invariants 1-4 hold in a state* $S$, *they hold in every state* $S' \geq S$.

*Proof.* Assume in a computation of BUILD-LIST* there is a state $S$ in which Invariants 1-4 hold, such that in the (direct) subsequent state $S'$ one of the Invariants 1-4 does not hold. First of all, check that none of the first four invariants can be invalidated because some node becomes leaving. Second, note that the first four invariants cannot become falsified due to a new INTRODUCE$(v, w)$ message: These messages are only sent by $w$ and only to a node $u \in Left(w) \cup Right(w)$ between closer to $w$ than $v$. Third, note that a new LINEARIZE$(v)$ message is sent by a node $u$ only upon receipt of an INTRODUCE$(v, w)$ message and only to the node $w$. Additionally, before sending that message, $v$ is added to $Left(u)$ or $Right(u)$. According to the first invariant, $R_s^+(id(w), u) \subseteq R_s(w)$ and $\min(id(v), id(w)) < id(u) < \max(id(v), id(w))$, implying that $R_s^+(id(w), v) \subseteq R_s(u)$. Thus, a newly sent LINEARIZE$(v)$ message also cannot falsify the first four invariants. Next, note that according to the protocol when a node $w$ sends a REVERSEANDLINEARIZEACK$(v, uniqueValue)$ to a node $u$, then $w = v$ and it makes sure that $uniqueValue$ is stored in $v.uniquevalues[u]$ (and we assume that $uniqueValue$ is only stored for $u$). Thus, sending such a message also cannot invalidate one of the first four invariants. Moreover, note that when a node $v$ sends a REVERSEANDLINEARIZE$(nodeList, uniqueValue)$ message to a node $u$ with $id(u) < id(v)$, then $v$ must have received a REVERSEANDLINEARIZEACK$(u, uniqueValue)$ message right before and $v$ must be leaving. Since Invariant 3 holds in $S$, this means that $u.uniqueValues[v] = uniqueValue$ and $v$ is the only node such that $u.uniqueValues[v] = uniqueValue$. In addition, when sending the message, $v$ added all nodes from $Left(v)$ or $Right(v)$ to $nodeList$, depending on whether $id(v) < id(u)$ or $id(v) > id(u)$, respectively. Thus, by Lemma 5.3 in state $S'$ if $id(v) < id(u)$ then $\forall\, x \in nodeList : id(x) < id(v)$ and if $id(v) > id(u)$ then $\forall\, x \in nodeList : id(x) > id(v)$. Furthermore, $R_s^+(id(u), v) = R_s(id(u), nodeList)$ holds and $v$ is the only node $v'$ with $u.uniqueValues[v'] = uniqueValue$ in $S'$. Besides, note that the $R_s^+(id(u), v) = R_s(id(u), nodeList)$ part of Invariant 4 for a node $v$ cannot be invalidated due to the addition of any node to the set $Right(v)$ or $Left(v)$ because $v$ is leaving and a leaving node never adds a member to $Right$ or $Left$. Any other addition of a node to a set $Right(x)$ or $Left(x)$ for another node $x$ adds this node to $R_s^+(id(u), v)$ and $R_s(id(u), nodeList)$ at the same time or not at all.

Thus, the only event that can invalidate one of the first four invariants is the removal of a node $y$ from a set $Right(x)$ or $Left(x)$ for a node $x$. This may only happen in a LINEARIZE$(y)$ action executed by a staying node or in a REVERSEANDLINEARIZE$(nodeList, uniqueValue)$ action. We consider both actions individually.

First of all, assume a LINEARIZE$(y)$ action is being executed by a staying node $w$ between $S$ and $S'$ and thus removed a node $y$ from $Right(w)$ or $Left(w)$. This can only happen if there was a LINEARIZE$(y)$ message in $w.Ch$ in $S$ for which, by definition of $S$, Invariant 2 holds. Thus, there is a node $u \neq y$ with $u \in Right(w) \cup Left(w)$ such that $\min(id(v), id(w)) < id(u) < \max(id(v), id(w))$ and $R_s^+(id(w), y) \subseteq R_s(u)$, implying that after the removal of $(w, y)$, $R_s^+(id(w), y) \subseteq$

$R_s(w)$ still holds: i.e., the removal of $y$ has not changed $R_s(w)$, nor any set $R_s(x)$ for any node $x$.

Now assume that a ReverseAndLinearize($nodeList, uniqueValue$) action has been executed in a node $u$ between state $S$ and $S'$. In this case, the corresponding message must have been in $u.Ch$ in $S$. Since in $S$ the first four invariants hold, by the fourth invariant, there must be exactly one node $v$ that is leaving with $u.uniqueValues[v] = uniqueValue$ and $R_s^+(id(u), v) = R_s^+(id(u), nodeList)$. W.l.o.g. assume that $id(u) < id(v)$ (note that in case $id(u) > id(v)$ and $u$ is leaving, no node is removed from or added to $Left(u)$ at all, but in this case, the invariants still hold, which is what we want to prove anyway). If $v \notin Right(x)$, no node is removed from or added to $Right(x)$ at all and the claim follows immediately. Thus, assume $v \in Right(x)$. In this case, $u$ removes $v$ from $Right(u)$ and adds $nodeList$ to $Right(u)$. Since $id(u) < id(v)$, $R_s^+(id(u), v) = R_s^+(id(u), nodeList)$, $R_s^+(id(u), v) \subseteq R_s(u)$ before the removal (by definition) and $v \notin R_s(u)$ (because $v$ is leaving), no node has been removed from or added to $R_s(u)$ after the action has been performed, implying that all four invariants still hold. □

Before we continue with the next invariant, we show the following lemma that will turn out to be very helpful. Intuitively, it tells that once a node $v$ can reach all staying nodes via a straight path that another node $x$ can reach via a straight path into the same direction, this will hold true in every subsequent state (note that this holds even if $x$ becomes leaving).

**Lemma 5.32.** *In every computation of* Build-List*, *if Invariants 1-4 hold in a state $S$ and for an arbitrary pair of nodes $v$ and $x$, $R_s^+(id(v), x) \subseteq R_s(v)$, then in every state $S' \geq S$, $R_s^+(id(v), x) \subseteq R_s(v)$.*

*Proof.* Assume in an arbitrary computation of Build-List* there is a state $S$ such that Invariants 1-4 hold and for an arbitrary pair of nodes $v$ and $x$, $R_s^+(id(v), x) \subseteq R_s(v)$, but in the (direct) subsequent state $S'$, $R_s^+(id(v), x) \subseteq R_s(v)$ does not hold. Without loss of generality, assume $id(v) < id(x)$. Note that this implies that for all $y \in R_s^+(id(v), x)$, $id(y) \geq id(x)$. We consider all possible reasons for why $R_s^+(id(v), x) \subseteq R_s(v)$ does not hold in $S'$. Obviously, neither the addition of a node to $R_s(v)$ nor the removal of a node from $R_s^+(id(v), x)$ can violate the claim. Note that if a node $z$ such that $id(z) > id(x)$ is added to $R_s^+(id(v), x)$, this happens because some node $y \in R_s^+(id(v), x)$ (implying $id(y) \geq id(x)$) added $z$ to $Right(y)$. However, since $y \in R_s(v)$ and $id(y) > id(v)$, $z$ is also added to $R_s(v)$ (by definition of this set). This yields that the only reason for the claim to be incorrect in $S'$ is that a (staying) node $z \in R_s^+(id(v), x)$ was removed from $R_s(v)$ but not from $R_s^+(id(v), x)$. In other words, in $S'$ there is a straight path from $x$ to $z$ into the "right direction", i.e., whose nodes have increasing identifiers, but there is no straight path from $v$ to $z$ (which, since $id(v) < id(x)$, would also need to go into the "right direction"). Recall that $id(z) > id(x)$ must hold in this case. We consider all possible cases for this having happened.

First, assume $z$ was removed from $R_s(v)$ because $z$ became leaving. Then $z$ was also removed from $R_s^+(id(v), x)$.

Second, assume that $z$ was removed from $R_s(v)$ due to a LINEARIZE($y$) action at a node $w \in R_s(v)$ between $S$ and $S'$ (note that according to the pseudocode, this happens only at staying nodes $w$). Note that since $z \in R_s^+(id(v), x) \subseteq R_s(v)$ in $S$ and $id(z) > id(x)$, $id(w) > id(v)$ and $id(y) > id(w)$ must hold in this case (if $id(w) < id(v)$ or $y$ was not in $Right(w)$ before, then this removal would be irrelevant since there is a "right direction" path from $v$ to $z$). Then, by the second invariant, there was a node $u \neq y$ with $u \in Right(w) \cup Left(w)$ and $R_s^+(id(w), y) \subseteq R_s(u)$, and $\min(id(y), id(w)) < id(u) < \max(id(y), id(w))$ in $S$. Note that this together with the fact that $id(w) < id(y)$ implies that $u \in Right(w)$ and $R_s^+(id(w), y) \subseteq R_s(w)$ even without the edge $(w, y)$. Thus, after $y$ is removed from $Right(w)$, $R_s^+(id(w), y) \in R_s(w)$ still holds. Since $w \in R_s(v)$ and $id(w) > id(v)$, this implies $R_s^+(id(v), y) \subseteq R_s(v)$: i.e., neither $y$ nor any other node $z$ in $R_s(y)$ was removed from $R_s(v)$.

Third, assume a staying node $z \in R_s^+(id(v), x)$ was removed from $R_s(v)$ but not from $R_s^+(id(v), x)$, due to a REVERSEANDLINEARIZE($nodeList, uniqueID$) action in a node $u$ such that $id(u) > id(v)$, and there is a straight path from $v$ to $u$, removing node $y$ from $Right(u)$. In this case, according to Invariant 4, $y$ is the unique node with $u.uniqueValues[y] = uniqueValue$, $y$ is leaving, and $R_s^+(id(u), y) = R_s^+(id(u), nodeList)$. Thus, when $y$ is removed from $Right(u)$ and $NodeList$ is added to $Right(u)$, no node is removed from $R_s(u)$, implying that no node is removed from $R_s(v)$, because of the straight path from $v$ to $u$ (note that this path cannot have been destroyed since the REVERSEANDLINEARIZE($nodeList, uniqueID$) action is the only action executed between $S$ and $S'$). Thus, the claim holds in every case. □

Using Lemma 5.32, we can prove the next sequence of lemmas:

**Lemma 5.33.** *In every computation of* BUILD-LIST*, *if Invariants 1-4 hold in a state $S$, then there is a state $S' \geq S$ such that Invariants 1-5 hold throughout $SUFFIX(S')$. Furthermore if Invariants 1-5 hold in some state $S''$, they hold in every state $S''' \geq S''$.*

*Proof.* Assume in an arbitrary computation of BUILD-LIST* that there is a state $S$ in which Invariants 1-4 hold. Lemma 5.31 implies that Invariants 1-4 hold throughout $SUFFIX(S)$. First, we will show that in $SUFFIX(S)$ existing messages conforming to Invariant 5 will never violate this invariant. Second, we will show that every new FORWARDPROBE() message sent during $SUFFIX(S)$ during the execution of TIMEOUT will conform to Invariant 5. Third, we will show that every new FORWARDPROBE() message sent during $SUFFIX(S)$ during the execution of the FORWARDPROBE() action upon receipt of a FORWARDPROBE() message that conforms to Invariant 5 does not violate Invariant 5. From these observations, it immediately follows that if Invariants 1-5 hold in some state $S''$, they hold in every state $S''' \geq S$ (note that there is no other occasion than the TIMEOUT and the

ForwardProbe() action that a ForwardProbe() message is sent). Fourth and last, to prove the first part of the claim, we argue that every ForwardProbe() violating Invariant 5 will cause only a finite number of other ForwardProbe() messages violating Invariant 5, which finishes the proof.

First, assume that there is a ForwardProbe($source, destID, Next, seq$) message that conforms to Invariant 5 in some state $S' \geq S$. Note that since $source.lseq[destID]$ is monotonically increasing and because of Lemma 5.32, it will do so in every state $S' \geq S$.

Second, assume a node $u$ sends a ForwardProbe($source, destID, Next, seq$) message during Timeout after $S$. According to the pseudocode, $u$ sends the message to $u$ itself, with $u = source$ and $Next = \{u\}$, which is why Invariants 5a) and 5b) hold for the new message. Also note that since $u.lseq[destID]$ is monotonically increasing and $seq = source.lseq[destID]$ in this state, if there was an admissible state with $source.lseq[destID] < seq$ with $v \in R_s(source, destID)$ for the node $v$ such that $id(v) = destID$, then this must have been a previous state. Note that $v \in R_s(source)$ implies $R_s^+(id(source), v) \subseteq R_s(source)$. By Lemma 5.32, $R_s^+(id(source), v) \subseteq R_s(source)$ must still hold when the message is sent, which, if $v$ is staying, implies $v \in R_s(source, destID)$. Thus, even Invariant 5c) holds for the new message. So all in all, Invariant 5 holds for the new message.

Third, assume a node $u$ sends a ForwardProbe($source, destID, Next, seq$) message after $S$ during the execution of the ForwardProbe() action upon receipt of a ForwardProbe() message that conforms to Invariant 5. Assume $id(source) \leq destID$ for the other case is analogous. Note that according to this pseudocode, $u$ must have received a ForwardProbe($source, destID, Next', seq$) message then. Furthermore, $u$ sends the ForwardProbe($source, destID, Next, seq$) message to the node with the minimum identifier in $Next$ and only if $id(u) \neq destID$, which is why Invariant 5a) holds for the new message. Moreover, recall that we assumed that Invariant 5 held for the ForwardProbe($source, destID, Next', seq$) message $u$ received. Thus, if there is a $v$ such that $id(v) = destID$ then $u \neq v$ and since $R_s^+(id(source), Next)$ and $R_s^+(id(source), Next')$ only differ in $u$ (since $Next = Next' \setminus \{u\} \cup Right(u)$), Invariant 5c) also holds for the new message. Furthermore, $R_s^+(id(source), Next') \subseteq R_s(source)$ implies $R_s^+(id(source), Next) \subseteq R_s(source)$ (recall that $Next = Next' \setminus \{u\} \cup Right(u)$), yielding the claim of Invariant 5b) for the new message. So all in all, Invariant 5 holds for the new message.

Fourth, note that according to the pseudocode, during the execution of ForwardProbe($source, destID, Next', seq$), the executing node $v$ adds nodes from $Right(v)$ to $Next'$ and sends the message to the node $u$ with the minimum identifier among all nodes from the resulting set $Next$, but only if this identifier is greater than $id(v)$, or it adds nodes from $Left(v)$ to $Next'$ and sends the message to the node with the maximum identifier among all nodes from the resulting set $Next'$, but only if this identifier is smaller than $id(v)$. In any case, according to the pseudocode only nodes whose identifier is between $v$'s and $destID$ are added. Define $range(Next) := |\max_x\{id(x) : x \in Next\} - \min_x\{id(x) : x \in Next\}|$. Since the new Forward-

PROBE($source, destID, Next', seq$) message is sent to a node in $Next$ and during every execution of FORWARDPROBE() only nodes whose identifier is between the current node and $destID$ are added to $Next$, we obtain by induction that eventually no new FORWARDPROBE() message is sent upon receipt of a FORWARDPROBE() message caused by the FORWARDPROBE($source, destID, Next', seq$) message. Thus, eventually all FORWARDPROBE() message violating Invariant 5 will be inactive and the proof is finished. □

**Lemma 5.34.** *In every computation of* BUILD-LIST*, *if Invariants 1-5 hold in a state $S$, then there is a state $S' \geq S$ such that Invariants 1-7 hold throughout $SUFFIX(S')$. Furthermore if Invariants 1-7 hold in some state $S''$, they hold in every state $S''' \geq S''$.*

*Proof.* Assume in an arbitrary computation of BUILD-LIST* that there is a state $S$ in which Invariants 1-5 hold. Lemma 5.33 implies that Invariants 1-5 hold throughout $SUFFIX(S)$. We will show that in $SUFFIX(S)$ existing messages conforming to Invariants 6 and 7 will never violate these invariants. Furthermore, we will show that all new PROBESUCCESS() and PROBEFAIL() messages sent during $SUFFIX(S)$ will conform to these two invariants. Thus, for a state $S'$ such that PROBESUCCESS() and PROBEFAIL() messages violating Invariants 6-7 that exist in $S$ have been received before $S'$, it holds that Invariants 1-7 hold throughout $SUFFIX(S')$. Also, from these observations, it immediately follows that if Invariants 1-7 hold in some state $S''$, they hold in every state $S''' \geq S$.

First of all, observe that a PROBESUCCESS($destID, dest$) message can only violate Invariant 6 if $id(dest) \neq destID$ or $dest$ is staying. Furthermore, a PROBEFAIL($destID, seq$) message can only violate Invariant 7 if there is a node $v$ with $id(v) = destID$ and $v$ is staying. Again, by Lemma 5.32 and because $R_s^+(id(y), x) \subseteq R_s(y)$ is equivalent to $x \in R_s(y)$ if $x$ is staying, and because $u.lseq[destID]$ is monotonically increasing for every $u$ and $destID$, existing messages conforming to Invariants 6 and 7 in some state $S'' \geq S$ cannot violate these invariants in any later state.

Therefore, we now consider the case of new PROBESUCCESS() and PROBEFAIL() messages being sent. Assume a new PROBESUCCESS($destID, dest$) message has been sent by $w$ to a node $u$. According to the protocol, this only happens in a FORWARDPROBE() action, when a FORWARDPROBE($source, destID, Next, seq$) message has arrived at $w = dest$ with $id(w) = destID$, $u = source$, and $w$ is staying. Thus, Invariant 5 b) implies $dest \in R_s(u)$. For the PROBEFAIL() messages, assume a node $w$ sends a PROBEFAIL($destID, seq$) message to a node $u$. Note that if there is no staying node with identifier $destID$, this message cannot violate Invariant 6. Thus, we assume that there is a staying node $v$ with $id(v) = destID$. Furthermore, assume $id(w) \leq destID$ (the other case is analogous). According to the protocol, $w$ sends the PROBEFAIL($destID, seq$) message only during the execution of a FORWARDPROBE() action caused by a FORWARDPROBE($source, destID, Next, seq$) that arrived at $w$ if $id(w) = destID$

and $w$ is leaving (which implies $w = v$ but contradicts that $v$ is leaving), or if $id(w) \neq destID$, $u = source$, $Next \setminus \{w\} = \emptyset$, and there is no $y$ in $Right(x)$ with $id(y) \leq destID$. Since we assumed $id(w) \leq destID$, Invariant 5a) implies $id(source) < id(w)$. This implies $v \notin R_s^+(id(source), Next)$. Invariant 5c) then yields that Invariant 7 holds for the ProbeFail($destID, seq$) message. □

We are now ready to prove the following result:

**Lemma 5.35.** *In every computation of* Build-List*, *if Invariants 1-7 hold in a state $S$, then there is a state $S' \geq S$ such that every state of $SUFFIX(S')$ is admissible. Furthermore if some state $S''$ is admissible, then every state in $SUFFIX(S'')$ is admissible.*

*Proof.* The idea of this proof is similar to that of Lemma 5.34. Assume in an arbitrary computation of Build-List* there is a state $S$ in which Invariants 1-7 hold. Lemma 5.34 implies that Invariants 1-7 hold throughout $SUFFIX(S)$. We will show that in $SUFFIX(S)$ existing Search() messages conforming to Invariant 8 will never violate this invariant and that every new Search() message sent during $SUFFIX(S)$ will conform to Invariant 8. Again, from these observations, it immediately follows that if there is an admissible state $S''$, then every state in $SUFFIX(S'')$ is admissible.

First, consider there is an existing Search($v, destID$) message in $u.Ch$ that conforms to Invariant 8 in some state $S'' \geq S$ but violates Invariant 8 in the state $S'''$ subsequent to $S''$. This can only be the case if $u \notin R_s(v)$ and $u$ is staying in $S'''$. Recall that $R_s^+(id(v), u) \subseteq R_s(v)$ is equivalent to $u \in R_s(v)$ if $u$ is staying. Thus, Lemma 5.32 implies $u \in R_s(v)$ yielding a contradiction.

Second, assume that a new Search($v, destID$) message is sent to a node $u$ after $S$. According to the protocol, such a message is only sent by $v$ itself and upon receipt of a ProbeSuccess($destID, u$) message. However, by assumption, Invariant 6 holds for this message, i.e., $id(dest) = destID$ and $dest \in R_s(v)$. Therefore, Invariant 8 holds for the newly sent Search($v, destID$) message. □

Now that we have shown that each computation that starts from an admissible state contains admissible states only, we show that every computation of Build-List* actually contains an admissible state. This is formalized by the following lemma:

**Lemma 5.36.** *In every computation of* Build-List* *there is an admissible state.*

*Proof.* Note that according to Lemma 5.6, every computation of Build-List* contains a state $S_1$ such that in $SUFFIX(S_1)$ all nodes that will eventually be leaving are inactive and note that these nodes do not perform any actions once they are inactive. Furthermore, note that according to the protocol a ReverseAndLinearize($nodeList, uniqueValue$) message is sent only if some node received a ReverseAndLinearizeACK($v, uniqueValue$) message. Moreover, a ReverseAndLinearizeACK($v, uniqueValue$) message can only be sent

if a node receives a REVERSEANDLINEARIZEREQ($dir$) message. Such a message is only sent from a leaving node. However, in $SUFFIX(S_1)$, no leaving node can send a message anymore. Thus, there is a state $S_2$ such that in $SUFFIX(S_2)$ the third and the fourth invariant always hold. Note that according to Lemma 5.24, there will be a state $S_3 \geq S_2$ such that in $SUFFIX(S_3)$ no INTRODUCE$(v, w)$ and no LINEARIZE$(v)$ messages will exist, implying the first two invariants hold throughout $SUFFIX(S_3)$. Since according to Lemma 5.31 the first four invariants hold throughout $SUFFIX(S_3)$, the sequence of Lemma 5.33, Lemma 5.34 and Lemma 5.35 implies that there will be an admissible state $S_4$. □

Note that Lemma 5.35 and Lemma 5.36 imply the following corollary:

**Corollary 5.37.** *In every computation of* BUILD-LIST*, *there exists a suffix in which every state is admissible.*

### 5.5.3. Proving the Correctness in Admissible States

In this section, we provide the last missing pieces required for the proof of Theorem 5.29. The lemmas and proofs of this section do not coincidentally closely resemble those of Section 4.4.4 from Chapter 4, since the same proof structure can be applied here. They mostly differ in some details and particularly the notation that needs to be different here to account for the possibility of leaving nodes. We begin with the following lemma:

**Lemma 5.38.** *In every computation of* BUILD-LIST*, *if a node $v$ initiates a* FORWARDPROBE*$(v, destID, Next, seq)$ message in an admissible state, then $v$ eventually receives either a* PROBESUCCESS*$(destID, dest)$ message for some node $dest$ or a* PROBEFAIL*$(destID, seq)$ message.*

*Proof.* For an arbitrary computation of BUILD-LIST*, let $S$ be an arbitrary admissible state. According to Lemma 5.35, every state of $SUFFIX(S)$ is admissible. As in the proof of Lemma 4.23, we use the potential function $\Psi(U, ID)$ defined as follows: $\Psi(U, ID) := \sum_{u \in U} n^{|id(u) - ID|}$, where $n$ is the total number of processes. Here, we additionally define $R_s^+(ID_1, U, ID_2) := \{u \in R_s^+(ID_1, U) : min(ID_1, ID_2) \leq id(u) \leq \max(ID_1, ID_2)\}$ for any identifiers $ID_1, ID_2$ and set of node references $U$. Observe that $u \in R_s^+(id(v), U)$ for an arbitrary set $U$ and nodes $u, v$ implies $u \in R_s^+(id(v), U, id(u)$.

First, note that if in $SUFFIX(S)$ a node $u$ receives a FORWARDPROBE$(v, destID, Next, seq)$ message and during the execution of the FORWARDPROBE() action executed upon receipt $u$ does not send PROBESUCCESS() or PROBEFAIL() message, then $u$ sends a FORWARDPROBE$(v, destID, Next', seq)$ message to some other node and $\Psi(Next', destID) < \Psi(Next, destID)$. This follows from the protocol and the fact that $u$ chooses $Next'$ as $Next \setminus \{u\}$ augmented by all neighbors whose identifier is between $id(u)$ and $destID$: i.e., it only adds neighbors $v$ such that $|id(v) - destID| < |id(u) - destID|$. Thus, $\Psi(Next', destID) < \Psi(Next, destID)$.

Also note that for every existing ForwardProbe$(v, destID, Next', seq)$ message, $\Psi(Next', destID)$ cannot increase and that $\Psi$ is lower bounded by 0.

Therefore, for an arbitrary ForwardProbe$(v, destID, Next, seq)$ message, we can inductively obtain that there must a node $w$ that receives a ForwardProbe$(v, destID, Next', seq)$ message but that does not send a new ForwardProbe() message upon receipt of that message. Instead, according to the protocol, it sends either a ProbeSuccess$(destID, w)$ message or a ProbeFail$(destID, seq)$ message to $v$. □

Making use of this lemma, we show the following intermediate result. Here, again, we say a message is *buffered at* $u$ if for a node $u$ this message is contained in $u.waitingFor[destID]$.

**Lemma 5.39.** *In every computation of* Build-List*, *if a staying node* $u$ *has a* Search$(u, destID)$ *message buffered at* $u$ *in an admissible state, then if* $u$ *does not become leaving, this message will eventually be delivered or dropped. In the former case, it will be delivered to the node* $w$ *with* $id(w) = destID$ *(which exists in this case). In the latter case, either there is no staying node with identifier* $destID$ *when the message is dropped or all previous* Search$(u, destID)$ *messages that were initiated before that message and that were buffered at* $u$ *during at least one admissible state have been or will be dropped as well.*

*Proof.* For an arbitrary computation of Build-List*, let $m$ be an arbitrary Search$(u, destID)$ message buffered at some staying node $u$ in an admissible state $S$. Furthermore, let $seq$ be the value of $u.lseq[destID]$ in $S$. If $u$ becomes leaving, the claim trivially holds, so assume $u$ remains staying forever. Recall that according to Lemma 5.35, every state of $SUFFIX(S)$ is admissible. Due to the protocol, node $u$ initiates a ForwardProbe$(u, destID, Next, seq)$ message every time it executes Timeout. According to Lemma 5.38, $u$ will eventually receive a ProbeSuccess$(destID, dest)$ or a ProbeFail$(destID, seq)$ message. Note that $u$ forwards or drops $m$ upon the first receipt of such message after $S$.

First, consider the case that the first such message that $u$ receives after state $S$ is a ProbeSuccess$(destID, dest)$ message. Invariant 6 yields $id(dest) = destID$ and $m$ will be sent to $dest$, according to the protocol.

Second, consider the case that the first such message that $u$ receives after $S$ is a ProbeFail$(destID, seq)$ message. According to Invariant 7, either no staying node $v$ with $id(v) = destID$ exists (in which case we are finished) or for every admissible state with $u.lseq[destID] < seq, v \notin R_s(u)$. Now consider an arbitrary earlier Search$(u, destID)$ message $m'$ that was buffered at $u$ during an admissible state. If $m'$ is still waiting at $u$ in state $S$, then $m'$ will be dropped together with $m$ when $u$ receives the ProbeFail$(destID, seq)$ message. Otherwise, assume for contradiction that immediately after some state $S' < S$, $m'$ was sent to a node $dest$ with $id(dest) = destID$. Since $m'$ was buffered at $u$ during at least one admissible state, by Lemma 5.35, $S'$ is admissible as well. According to the protocol, the fact that $m'$ was sent to $dest$ requires that there was a ProbeSuccess$(destID, dest)$

message in $u.Ch$ in $S'$. By Invariant 6, *dest* was leaving in $S'$, or $dest \in R_s(u)$ in $S'$. In the former case, we are done because *dest* will then also be leaving in every later step and, in particular, when $m$ is dropped. In the latter case, note that $u$ increased $u.lseq[destID]$ when the first SEARCH() message was initiated after $S'$: i.e., before $S$. Since the sequence numbers are monotonically increasing, $S'$ is a state with $u.lseq[destID] < seq$. Thus, Invariant 7 of the PROBEFAIL$(destID, seq)$ message implies $dest \notin R_s(u)$ in state $S'$, yielding a contradiction. This finishes the proof. □

In the following, to simplify the description, we say a node $u$ is *always staying* in a computation $C$ if it is staying initially in $C$ and never becomes leaving.

We continue with the following result that represents the last lemma on the way to prove Theorem 5.29 (which basically uses the same ideas as Lemma 4.24 from Chapter 4):

**Lemma 5.40.** *In every computation $C$ of* BUILD-LIST*, *for every two always staying nodes $u$ and $v$ such that $v \in R_s(u)$ in some admissible state $S$, there is a state $S' \geq S$ such that all* SEARCH*$(u, id(v))$ messages initiated in $S'$ and all subsequent states will be delivered to $v$.*

*Proof.* For an arbitrary computation of BUILD-LIST*, let $u$ and $v$ be two always staying nodes in $C$ and assume $v \in R_s(u)$ in an admissible state $S$. Note that according to the pseudocode and Lemma 5.39, there will be a state $S' \geq S$ in which $u.WaitingFor[id(v)]$ is empty because all SEARCH$(u, id(v))$ messages buffered at $u$ during $S$ have been sent to their destination or dropped right before (if there were any). Let $seq'$ be the value of $u.lseq[id(v)]$ in $S'$. Consider the first SEARCH$(u, id(v))$ message initiated after $S'$. Since INITIATENEWSEARCH$(id(v))$ is the only action that adds elements to $u.WaitingFor[id(v)]$, $u.WaitingFor[id(v)] = \emptyset$ holds when this message is initiated. According to the pseudocode (c.f. Listing 5.2), $u.lseq[id(v)]$ will be increased by one at that time: i.e., its new value will be $seq'' = seq' + 1$. Now consider an arbitrary SEARCH$(u, id(v))$ message $m$ initiated after $S'$. By Lemma 5.39 $m$ will be delivered or dropped. Assume for contradiction it is dropped. According to the pseudocode (c.f. Listing 5.2), this requires $u$ to receive a PROBEFAIL$(destID, seq)$ message with $destID = id(v)$ and $seq \geq u.lseq[id(v)]$. Note that as argued before, at that time $u.lseq[id(v)] \geq seq'' > seq'$. Thus, Invariant 7 for the PROBEFAIL$(destID, seq)$ yields a contradiction to $v \in R_s(u)$ in $S$. All in all, we have that $m$ is delivered correctly. □

Armed with these lemmas, the proof of Theorem 5.29 is straightforward. Before providing its proof, we restate the theorem:

**Theorem 5.29.** BUILD-LIST* *admissible-message satisfies monotonic searchability according to* SEARCH*.

*Proof.* Consider an arbitrary computation $C$ of BUILD-LIST*. Corollary 5.37 yields that $C$ contains an admissible state and that the computation suffix starting from

the first admissible state $S$ solely consists of admissible states. Thus it remains to be shown that the protocol satisfies monotonic searchability in $SUFFIX(S)$.

Consider an arbitrary SEARCH$(u, destID)$ message $m$ initiated in a node $u$ after $S$ that is successfully delivered to the node $v$ with $id(v) = destID$. If $u$ or $v$ become leaving at some point in time, there is nothing to prove. Thus, assume $u$ and $v$ are always staying. Assume for contradiction that there is another SEARCH$(u, destID)$ message $m'$ initiated after $m$ that is dropped. Note that both $m$ and $m'$ were buffered at $u$ during at least one admissible state. Lemma 5.39 implies that since $m'$ is dropped, $m$ must have been dropped as well, which represents a contradiction. Thus, every SEARCH$(u, destID)$ initiated after $m$ is delivered as well.

Last, note that in legal states (i.e., in the line topology) for every pair of staying nodes $u$ and $v$, $v \in R_s(u)$. Thus, Lemma 5.40 yields that BUILD-LIST* fulfills the non-triviality property, which completes the proof of Theorem 5.29. □

PART

# III

# Relays: A New Model for the Interconnection of Nodes

The third main part of this thesis introduces the relay model and answers the following questions: *What does a reasonable model not requiring oracles to solve the finite departure problem look like? How can this model be implemented in a self-stabilizing fashion, assuming only a reliable link-layer?* To answer these questions we introduce the relay model for the interconnection of nodes. We describe its application programming interface, propose a pseudo-code level implementation that assumes only a reliable link-layer and prove that it is indeed self-stabilizing. To motivate that the proposed model is reasonable, we show that it is universal in the sense that arbitrary graphs can be transformed into arbitrary other graphs. Thereby, once again, a set of graph transformation primitives based on the primitives in $\mathcal{IDFR}$ plays a significant role. In addition, we show that existing protocols for well-established models can be transformed into the relay model. Furthermore, we show a general approach to transform protocols such that they solve the $\mathcal{FDP}$ in the relay model.

CHAPTER 6

# The Relay Model and Its Self-Stabilizing Realization

In this chapter, we describe a novel model for the interconnection of nodes, which permits a solution to the finite departure problem. The development of this model was driven by two insights gained from the proof that the $\mathcal{FDP}$ is unsolvable in the standard model without oracles: First, the standard assumption used in overlay networks research that a node may freely pass knowledge about its neighbors to any one of its neighbors has the effect that a node $v$ cannot locally decide whether $v$ is critical for the connectivity of the network or not, simply because $v$ does not have any control on and thereby potentially incomplete knowledge about its incoming connections (i.e., the set of nodes having a reference of $v$). Second, when assuming asynchronous communication, where messages may have arbitrary finite delays, a node $v$ may not know whether messages carrying critical connectivity information are still on their way to $v$. Our model overcomes these issues by managing connections through relays. Each node can query each of its relays whether it still has incoming connections and an outgoing connection is closed only if all messages sent via the corresponding relay have already been delivered. This way, the $\mathcal{FDP}$ is solvable without the use of oracles, as we will show.

At first glance, one might suspect that our model is a simple implementation of an oracle used to solve the $\mathcal{FDP}$. However, our model has several advantages that qualify it to be of broader interest. First of all, as we show, it can be implemented in a self-stabilizing fashion. Furthermore, it is universal in the sense that one can transform arbitrary initial graphs into arbitrary final graphs in this model. Moreover, existing protocols can be transformed such that they can be executed in the relay model as well. Besides, our model allows nodes to grant and revoke access rights, which opens up many new possibilities.

The main results of this chapter have previously appeared in the following publication:

> Christian Scheideler and Alexander Setzer. **Relays: A New Approach for the Finite Departure Problem in Overlay Networks**. In: *Proceedings of the 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Tokyo, Japan, 2018 [SS18]

**Outline of This Chapter** We start this chapter in Section 6.1 with a definition of the communication model used in this part of the thesis and a formal statement of the problem considered here. After that, in Section 6.2, we describe our new

interconnection model, called the *relay layer*, as well as its commands, variables, internal operation (relying on the reliable link layer only) and properties. In Section 6.3, we then prove that it actually fulfills these claimed properties. In Section 6.4, on the basis of the primitives for the manipulation of graphs defined in Chapter 2, we define a set of relay primitives that we prove to maintain weak connectivity, to be universal and to allow transferring existing protocols to our model. Last, in Section 6.5 we present a general approach to solve the $\mathcal{FDP}$ with relays.

## 6.1. Communication Model and Problem Statement

In this section, we formally define the communication model used in this part of the thesis (Section 6.1.1). Furthermore, we define the desired properties and benefits that the relay model should exhibit (Section 6.1.2).

### 6.1.1. Communication Model

We assume that there is a reliable link layer that transmits messages from nodes to other nodes on the basis of the identifier of the target node contained in the message. More specifically, each node specifies a set of system-based variables, called *buffers*, containing messages to be sent to other nodes. The identifier of the respective target node is stored inside the message. We assume that the link layer may take an arbitrary but finite amount of time to process a message that was put into one of these variables, but these messages never get lost.

In this model, we say that a message $m$ requests to call an action $A$ at some node $u$ if $m$ corresponds to $A$ and $m$ has been transmitted to $u$ by the link layer but has not been processed yet. When $A$ is executed because it was enabled due to $m$, we say that $m$ is processed. We assume the link layer makes sure that every transmitted message will eventually be removed from the buffer it was taken from after it has been processed by the receiver.

There are no resources available beyond the nodes and the link layer as specified above (such as shared storage or a gateway), so the nodes rely entirely on themselves and the link layer in order to handle certain tasks. This implies that there is no way for two disconnected components of nodes to connect to each other.

We specify more details on how the nodes are interconnected with each other once we introduce relays in Section 6.2. As in the communication model used in Part II, we do not allow the presence of connections to nonexistent nodes.

### 6.1.2. Problem Statement

Since the standard interconnection model considered in the previous two chapters of this thesis does not permit a solution to the $\mathcal{FDP}$ without oracles, the goal of this chapter is to devise a new interconnection model, called the *relay model*, in which a self-stabilizing, local-control protocol can solve the $\mathcal{FDP}$ without oracles. The

relay model should make as few assumptions as possible, with the only exception of sourcing out the difficulties involved with message transport to the link layer. In particular, assuming a reliable link layer, the relay model should be able to recover from faults in the variables of the nodes in a self-stabilizing fashion. This means that after a finite amount of time, all operations specified by the interconnection model are guaranteed to behave in the specified way. In addition to permitting a solution to the $\mathcal{FDP}$, the relay model should be universal in the sense that it is possible to transform any weakly-connected topology into any other weakly-connected topology, which is important to make it a useful interconnection model for overlay networks. In the course of this, we show that existing protocols for the standard interconnection model can be transformed into protocols for the relay model. As we will see, the relay model will also allow nodes to grant and revoke access rights (i.e., a node can control and restrict its incoming connections), which offers further widespread applications of the model.

## 6.2. The Relay Layer

We assume that all connections between nodes happen through *relays*, which are managed by a *relay layer*. Each node $v$ is assumed to interact with its own separate relay layer $RL(v)$ (so that it is clear which relay is owned by which node) and $RL(v)$ is required to reside at the same machine as $v$ so that interactions between $v$ and $RL(v)$ are local. Whenever a message needs to be sent to $v$, it has to go through $RL(v)$. Each $RL(v)$ has a globally unique address, in short $RID$, that depends on the address of its machine so that messages can be sent to it from any other relay layer that knows its $RID$.

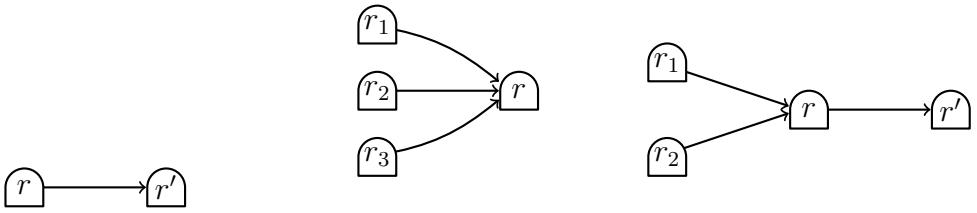
In addition to the buffers of the relays that will be specified below, every relay layer $RL(v)$ has a local buffer $RL(v).Buf$ that is used for the internal communication between relay layers: Every $RL(v).Buf$ is expected to consist of pairs $(targetRID, message)$ in which $targetRID$ is the $RID$ of the relay layer that $message$ is sent to by the link layer. This buffer can only be used for internal messages: i.e., messages used by the protocol for the operation and self-stabilization of the relay layer (which we will define later on). Any other type of message in this buffer will be ignored.

The relay layer of the entire system is the set of relay layers over all of its nodes.

### 6.2.1. High-Level Description

Before we start with the formal details of the relay layer and its commands, we give a high-level description of which relays are and how they are interconnected.

The basic concept of a relay is similar to that of a network socket: It is an endpoint for sending or receiving messages and it is non-transferably owned by exactly one node. In contrast to a network socket, however, a relay can have both incoming connections from other relays as well as an outgoing connection to some relay (some examples of such relay connections are shown in Figure 6.1).




(a) Relay $r$ has an outgoing connection to $r'$ and no incoming connection.

(b) Relay $r$ has three incoming connections (from relays $r_1$, $r_2$, and $r_3$) and no outgoing connection.

(c) Relay $r$ has two incoming connections (from relays $r_1$, and $r_2$) and one outgoing connection (to relay $r'$).

Figure 6.1.: Examples of relay connections.

The simplest form of a relay is a *sink relay*. A sink relay is a relay without any outgoing connection. It can have any number of incoming connections, including zero. A sink relay is the only kind of relay that can be created explicitly. To do this, a node $v$ can call the relay layer command **new Relay**. $RL(v)$ then creates a new sink relay $r$ owned by $v$ (we also say that the relay is owned by $RL(v)$, which is equivalent since there is exactly one relay layer $RL(v)$ for $v$). Whenever a message is sent via a sink relay $r$, $RL(v)$ puts this message into $r.Buf$. Any message in $r.Buf$ is then delivered to the node owning $r$. This alone does not make a sink relay very useful, since only the node owning $r$ could directly send a message via $r$. However, whenever a relay $r$ receives a message via one of its incoming connections, the relay layer puts the message into $r.Buf$ as well. As explained before, this has the effect that every message received by a sink relay is delivered to the node owning that sink relay.

A relay that has an outgoing connection is called a *non-sink relay*. Note that a relay cannot have more than one outgoing connection. The endpoint of the outgoing connection of a non-sink relay is another (sink or non-sink) relay. In principle, if a message $m$ is sent via a non-sink relay $r$, it is put into $r.Buf$, as was the case for a sink relay. For a non-sink relay $r$, however, the underlying link layer delivers $m$ to the endpoint of $r$'s outgoing connection, $r'$. As mentioned before, the relay layer owning $r'$ acts as though $m$ was directly sent via $r'$: it puts $m$ into $r'.Buf$. If $r'$ is a sink relay, this means that $m$ will be delivered to the node owning $r'$. Otherwise, $m$ will be sent to the endpoint of the outgoing connection of $r'$.

In a legal system state, each sequence of relay connections is finite, ending with a sink relay. Thus, every message sent via a relay will eventually be received by some node. Note that since a relay can have at most one outgoing connection but an arbitrary number of incoming connections, the connections between relays in the system form a forest of *relay trees* (see Figure 6.2 for an example of a relay tree and an illustration of the following notions). A relay tree $T$ has interesting properties. The root of $T$ is a unique sink relay $r$. The node owning $r$ receives all messages sent by any of the relays in the tree. Therefore, we say that for an

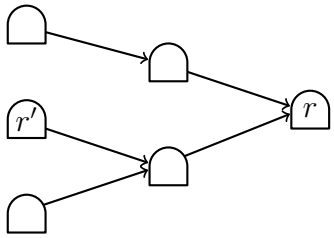



Figure 6.2.: A relay tree $T$. $r$ is the root of $T$ and the sink of every relay in $T$. The level of $r'$ in $T$ is two.

arbitrary relay $r'$ in $T$, $r$ is the *sink* of $r'$. Furthermore, we can define the *level* of a relay by its distance to the root: a sink relay has a level of zero, the relays that share an outgoing connection with $r$ have a level of one, and so on. As a side note, we mention that for technical reasons, all nodes in $T$ also store the RID of $r$.

Each relay can be *alive* or *dead*. A relay $r$ that is dead is prepared for deletion: It does not have any incoming connections, nor does it accept any new messages (i.e., the node owning $r$ could not send any message via $r$). It continues to exist, though, as long as $r$ is still needed. This may be for two reasons, one of which is that there are messages in $r.Buf$ that have not been transmitted by the link layer yet. The other reason is more technical and will be explained later on. A newly created relay is initially alive and can become dead only when the node owning it requests to **delete** it. A dead relay cannot become alive again.

The relay layer offers some specific commands for relays, in addition to the **delete** command we already mentioned. For example, one can check whether the relay has incoming connections via **has-incoming**. **direct** returns whether a relay is *direct*, meaning that it is either a sink relay or has an outgoing connection to a sink relay (i.e., its level is at most one). It is also possible to check whether a relay is a sink relay via **is-sink**. To check whether a relay still exists, one can call **dead** on a reference for a relay: it returns true if and only if that relay does not exist anymore or is dead. It is also possible to check whether two relays have an outgoing connection to the same relay (or are both sink relays) via **same-target**. Furthermore, sending a message is achieved via **send**. We mention that there is one additional command, **merge**, which can be used to *merge* several relays if all of them have an outgoing connection to the same relay and no incoming connections. Simply put, this means that the merged relays themselves are removed and a single new relay is created with the same outgoing connection. This simple description, however, lacks some important details and does not reveal the full purpose of this command, which can be understood later on when the full specification of the relay layer has been presented.

So far, we have described relays, some of their properties and their commands. For the description, we used the notion of a *connection* between two relays without further specifying its meaning. We now close this gap. Recall that the system

model only specifies connections between nodes. Specifically, a node $u$ is connected to another node $v$ if and only if $u$ knows $v$'s identifier. Relay connections, on the other hand, are maintained by the relay layer and in the following manner: each relay has a globally-unique identifier (called ID in the following). To achieve this, it consists of two parts: the RID of the relay layer owning that relay (recall that it is globally unique) and a locally unique identifier. The fact that the RID of the relay layer owning a relay $r$ is part of $r$'s ID has another benefit: it enables relay layers of other nodes to send messages to $r$ when knowing $r$'s ID. For every non-sink relay $r$, the relay layer stores the ID of the relay that $r$ is connected to in a variable attached to $r$. This way, messages put into $r.Buf$ can be transmitted as specified above. In order to fulfill the desired access-control requirements, connections are authorized via keys: each non-sink relay stores a key for its outgoing connection (more precisely, it stores a set of keys, but since the reason for this is not obvious at this point and to simplify the description, we here assume that there is only a single key). Moreover, each relay with incoming connections stores a set of "authorized" keys that may be used to send messages to this relay. When a message is sent via some relay $r$ to some relay $r'$, the key that $RL(r)$ stores for the connection of $r$ to $r'$ is attached to the message. Upon receipt of the message, $RL(r')$ checks whether the key attached to that message is stored in its set of authorized keys. If so, it puts the message into $r'.Buf$. Otherwise, it communicates with $RL(r)$ to cause $r$ to be deleted (messages sent via $r$ would not be able to reach the sink anyway then). This approach is the key strategy to enable access-control: if a connection only required the endpoint's ID, it could still be duplicated and distributed among many nodes. With the keys in place, although a key could be duplicated as well, a user could *revoke* a key that has been compromised (and that is used for an attack, for example). Since this would still only be reactive, for every incoming connection of a relay the relay layer stores the RID of the other endpoint of that connection, along with each key, and accepts messages sent from that RID only.

One important question has not been answered so far: how are relay connections created? We do not consider this question for the initial setup (this is beyond the scope of this thesis and should be investigated individually) and assume that the system starts from a state such that the graph that contains an edge $(u, v)$ if an only if there is a connection between two relays $r_u$ and $r_v$ owned by $u$ and $v$, respectively, is weakly connected. As we will show in the course of this chapter, in the non-self-stabilizing setting (in which we assume that the initial state is a legal state: i.e., there are no corrupted variables and messages) this requirement on the initial state is enough to be able to transform the system arbitrarily: i.e., to form any desired forest of relay trees (this is the main contribution of Section 6.4). The basic idea for the creation of connections is that references to relays can be sent in messages. As soon as a sink $r$ receives a message with a relay reference to some relay $r'$ in it, $RL(r)$ automatically creates a new relay that has $r'$ as its endpoint (and the relay layers take care that there are suitable keys for the connections as well).

This completes the high-level description of the relay layer. Accurate and formal

descriptions of the relay layer, as well as the variables, commands and actions are provided in the next subsections.

### 6.2.2. Relays and Relay Graphs

We now formally define the variables of a relay. For a node $v$, $RL(v)$ maintains the following variables for each relay $r$:

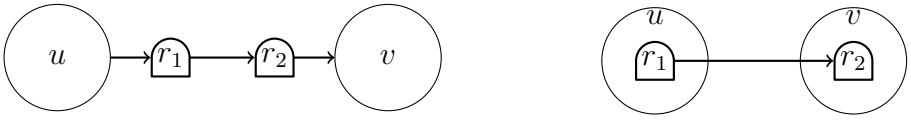
- $r.ID$: the globally-unique identifier of relay $r$ (containing the RID of $RL(v)$ so that messages can be sent to $r$ when knowing its ID),
- $r.state$: is either *alive* or *dead*,
- $r.out$: stores a $(Key, ID)$ pair, where $Key$ is a set of keys and $ID$ is the ID of the target of the outgoing connection (if $ID = \perp$ then $r$ is a *sink relay*: i.e., messages sent to $r$ are forwarded to the node owning $r$),
- $r.level \in \mathbb{N}_0$: stores the distance of $r$ (in hops) to the sink relay reached via its outgoing connection (there is always a unique such sink relay, see below),
- $r.sinkRID$: stores the RID of the *sink* of $r$: i.e., the RID of the relay layer of the node that will receive messages sent via $r$,
- $r.In$: a set of triples of the form $(key, RID, \perp)$ or $(key, \perp, r')$ for some relay $r'$, where $key$ is a globally unique key (depending on the RID of $r$'s relay layer), $RID$ specifies the address of the relay layer that can send messages to $r$ via $key$, and $r'$ is a relay via which $key$ was supposed to be forwarded; depending on the form, $key$ is a *confirmed* or *unconfirmed* key,
- $r.Buf$: stores all messages that the link layer should send to the relay layer whose RID is contained in $r.out.ID$ if $r.out.ID \neq \perp$, or to $v$ if $r.out.ID = \perp$.

Note that we assume all buffers (i.e., $RL(v).Buf$ and $r.Buf$ for every relay $r$) to be insert-only: i.e., only the link layer can remove a message from them. Furthermore, we assume all IDs in the system to be valid: i.e., for every ID in the system the corresponding RID belongs to an existing node (it would be possible to lift this assumption by introducing another oracle or by giving more power to the underlying link layer, but this is beyond the scope of this work).

The relay connections can be represented by a *relay graph*.

**Definition 6.1** (Relay Graph)**.** *Given any system state $S$, the* relay graph $G = (V, E)$ *of $S$ is a directed graph that is defined as follows: $V = R \cup P$, where $R$ is the set of relays and $P$ is the set of nodes. $E = E_P \cup E_{Ch}$, where $E_P$ is the set of all* explicit *edges and $E_{Ch}$ is the set of all* implicit *edges. $E_P$ contains an edge $(v, w)$ whenever*

1. *$v \in P$, $w \in R$, and $w$ is owned by node $v$,*




(a) Normal visualization of a relay graph according to the definition.

(b) Simplified visualization: relays are drawn inside the nodes they are owned by.

Figure 6.3.: Example of a relay graph. $u$ and $v$ are nodes, whereas $r_1$ and $r_2$ are relays. In later figures, we will use the simplified visualization (right image) only.

2. *$v \in R$, $w \in R$, and relay $v$ has an outgoing connection to relay $w$ (i.e., $v.out.ID = w.ID$), or*

3. *$v \in R$, $w \in P$, and relay $v$ is a sink relay of node $w$ (i.e., $v.out.ID = \perp$).*

*$E_{Ch}$ contains an edge $(v, w)$ whenever $v \in R$, $w \in R$ and a reference to $w$ is contained in the parameter list of a message in $v.Buf$. Thus, while explicit edges can be used to send messages, implicit edges cannot be used to send messages yet.*

Figure 6.3 shows an example of a relay graph. As we will see, the definition of legal states will require the relay graph to be cycle-free.

### 6.2.3. Commands

Whenever a node holds a reference to a relay $r$, which we denote by $\hat{r}$, we assume that it is a "dark" reference: i.e., the variables of the relay cannot be accessed by the node. However, the reference can be used by the node to call a number of commands offered by the relay layer (in the following, we assume that all relays mentioned below are *owned* by the calling node: i.e., they are or have been created for it by its relay layer — relays not owned by the calling node will be ignored):

- **new Relay**: returns a reference to a new sink relay $r$ with a globally unique identifier $r.ID$ containing $RID$, $r.state = alive$, $r.out = (\{\}, \perp)$, $r.level = 0$, $r.sinkRID = RID$, $r.In = \{\}$, and $r.Buf = \{\}$, where $RID$ is the RID of the executing relay layer.

- **delete** $\hat{r}$: immediately completely removes $r$ if $r$ is a sink relay. Otherwise it prepares $r$ for deletion, in a sense that the relay layer sets $r.In = \{\}$ and $r.state = dead$. This has the effect that $r$ will not accept any further messages, but $r$ will still continue to deliver the messages in $r.Buf$. $r$ is removed completely by the relay layer once $r.Buf$ is empty and all relay keys sent via $r$ have been confirmed or deleted (see below).

- **merge**($R$): only executes if for all relays $r \in R$, $r.state = alive$, $r.out.ID$ is equal to some common $ID$, $r.level$ is equal to some common $\ell$, $r.sinkRID$ is

equal to some common $sinkRID$, $r.In = \{\}$, and for every $key \in r.out.Key$, there is no other relay $r'' \notin R$ owned by the same node such that $key \in r''.out.Key$. In this case, the relay layer creates a new relay $r'$ with a new and globally unique $r'.ID$, $r'.state = alive$, and $r'.out = (Key, ID)$ with $Key = \bigcup_{r \in R} r.out.Key$, $r'.level = \ell$, $r'.sinkRID = sinkRID$, $r'.In = \{\}$, and $r'.Buf = \bigcup_{r \in R} r.Buf$. Additionally, for all relays $r \in R$, all relays $r''$ and all $(key, \bot, r) \in r''.In$, it replaces $(key, \bot, r)$ by $(key, \bot, r')$ in $r''.In$. Also, all relays in $R$ are immediately removed completely. A reference to $r'$ is returned back to the node. If one of the conditions above is not satisfied, **merge** does nothing.

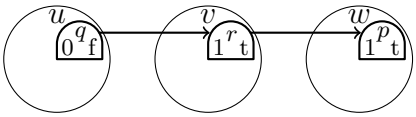
- **getRelays**: returns (references to) the current set of all relays owned by $v$ that are still alive.
- **has-incoming($\hat{r}$)**: returns true if and only if $|r.In| > 0$.
- **direct($\hat{r}$)**: returns true if and only if $r.level \leq 1$.
- **is-sink**($\hat{r}$): returns true if and only if $r.level = 0$.
- **dead($\hat{r}$)**: returns true if and only if $r$ does not exist anymore or $r.state = dead$.
- **same-target($\hat{r_1}, \hat{r_2}$)**: returns true if and only if $r_1.out.ID = r_2.out.ID$.
- **send($\hat{r}, action(parameters)$)**: if $r$ is still alive, this adds a message of the form TRANSMIT$((key, RID, r.out.ID), action(parameters'))$ for some arbitrary $key \in r.out.Key$ to $r.Buf$ (where $RID$ is the RID of the executing relay layer and $parameters'$ is an adapted form of $parameters$ explained below). The surrounding TRANSMIT is used as a wrapper to indicate that this message is not an internal type of message.

Figure 6.4 gives examples of the uses of these commands.

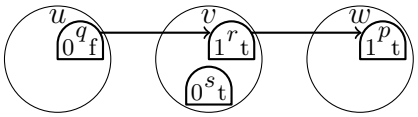
If a node $v$ executes **stop**, $v$ becomes inactive and $RL(v)$ immediately deletes all sink relays and from then on periodically deletes all relays $r$ such that $r.In = \emptyset$, $r.Buf = \emptyset$ and all keys sent via $r$ have been confirmed or deleted (i.e., there is no relay $r'$ such that $(key, \bot, r) \in r'.In$). $RL(v)$ continues to exist until all relays have been deleted, after which it shuts down. As we will prove, we highlight that protocols can prevent relay layers from existing forever by making sure that every indirect relay (i.e., every relay $r$ such that $direct(\hat{r}) = false$) is eventually closed.

Note that the fact that **merge** can be used to merge relays is the reason for why the variable $r.out.Key$ of a relay $r$ has to be a set instead of only a single value: The merge could occur in an illegal state at which one of the merged relays may store a correct key while another one does not. At this point it is not clear which one to choose.

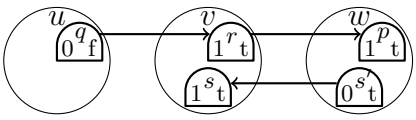
For convenience, in the following we will use $RL(r)$ to denote the relay layer that owns a relay $r$, $RID(ID)$ to denote the RID contained in $ID$, $RID(u)$ to denote the RID of $RL(u)$ for a node $u$ and $RID(r)$ to denote the RID of $RL(r)$.



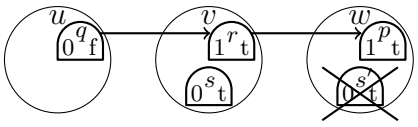

(a) Initial situation. $u$ owns relay $q$, $v$ owns relay $r$ and $w$ owns relay $p$. By definition, $r$ and $p$ are direct relays, whereas $q$ is not.




(b) Situation after $v$ has executed **new Relay**. $v$ has an additional (sink) relay $s$. By definition, $s$ is a direct relay.




(c) Situation after $v$ has executed $\text{SEND}(\hat{r}, action(\hat{s}))$ for some action *action*. $RL(w)$ has created a new relay $s'$ with an outgoing connection to $s$.




(d) Situation after $w$ has executed **delete** $\hat{s}'$. $s'$ is marked as dead, $s.In$ has been updated (as $s$ no longer has an incoming connection) and the connection from $s'$ to $s$ has been removed.

Figure 6.4.: Example with three nodes $u$, $v$, and $w$. The characters inside a relay $r$ denote (from left to right) $|r.In|$, the ID of $r$ and whether $r$ is a direct relay. The arrows indicate outgoing connections of relays.

### 6.2.4. Message Processing and Action Handling

All messages that can be sent by a node $v$ are required to be remote method invocations of the form $action(parameters)$ (otherwise, they will be ignored by $RL(v)$). More precisely, a node $v$ calls $send(\hat{r}, action(parameters))$ to ask $RL(v)$ to send out a message via $r$. For simplicity, we assume $parameters$ to consist of a sequence of objects, some of which are relay references, and all other objects do not contain any relay reference at all. We assume that each action has a fixed number of parameters and specifies which of its parameters are relay references. When $send(\hat{r}, action(parameters))$ is called for an alive relay $r$, there are two possibilities: If $r$ is a sink, i.e., $r.out.ID = \perp$, then $action(parameters)$ is put into $r.Buf$ such that the node owning $r$ will receive $action(parameters)$. Otherwise, $RL(r)$ for every relay reference $\hat{s}$ contained in $parameters$ creates a new globally unique key $key$, inserts $(key, \perp, r)$ into $s.In$ and replaces $\hat{s}$ by the quadruple $(key, s.ID, s.level + 1, s.sinkRID)$. We refer to these quadruples by the term *relay parameter*: the first entry of which is called its *key*, the second its *id*, the third its *level* and the fourth its *sinkRID*. Furthermore, we assume that there is a part of each generated key that depends on the generating node and can be used to check whether a key $key$ was generated by a node $u$, in which case we say *key belongs to u*. Let the list of parameters resulting from the replacements be $parameters'$. Then, $RL(r)$ puts a $\text{TRANSMIT}((r.out.Key, RID(r.ID), r.out.ID), action(parameters'))$ message into $r.Buf$. The triple $(r.out.Key, RID(r.ID), r.out.ID)$ is also called the *header*

of the message. We also say that a message *uses key msgkey* if *msgkey* is an element of the first entry of that message's header. The pseudocode of the SEND() action can be found in Listing 6.1.

Listing 6.1: Pseudocode of **send**

```
1  send(r̂, action(parameters))
2     if r.state = alive then
3     |  if r.out.ID = ⊥ then
4     |  |  r.Buf := r.Buf ∪ {action(parameters)}
5     |  else // r is not a sink relay
6     |  |  parameters' := parameters
7     |  |  let ŝ_1, ..., ŝ_k denote elements of parameters' that are relay references
8     |  |  for every i ∈ {1, ..., k} do
9     |  |  |  create a new globally unique key key
10    |  |  |  s_i.In := s_i.In ∪ {(key, ⊥, r)}
11    |  |  |  s'_i := (key, s_i.ID, s_i.level + 1, s_i.sinkRID)
12    |  |  |  replace ŝ_i in parameters by s'_i
13    |  |  // let key be arbitrary such that key ∈ r.out.Key
14    |  |  r.Buf := r.Buf ∪ {TRANSMIT(((r.out.Key, RID(r.ID), r.out.ID),
15    |  |     action(parameters')))}
```

We assume that the link layer for every relay $r$ eventually processes every message in $r.Buf$ without changing its contents. The link layer makes sure that every message $m' \in r.Buf$ for a relay $r$ is either processed by the node $v$ owning $r$, in case that $r.outID = \perp$, or successfully delivered to the node whose relay layer has the RID contained in $r.out.ID$. After the link layer has processed a message $m'$ in $r.Buf$ for a relay $r$, it removes $m$ from $r.Buf$. Recall in the following that relay layers can also communicate internally by putting $(targetRID, message)$ pairs into $RL.Buf$.

As we have seen, the header of a message is chosen in a specific way. The purpose of this header is to enable the receiving relay layer of a message to check whether this message was allowed to be sent to that relay. We therefore introduce the following definition:

**Definition 6.2** (Valid message header)**.** *A message $m$ of the form $((Keys, senderRID, outID), action(parameters))$ is said to have a* valid *header for relay $r$ if and only if*

*(i)* $r.ID = outID$ *and*

*(ii) there is a* $key \in Keys$ *such that* $(key, senderRID, \perp) \in r.In$ *or* $(key, \perp, r') \in r.In$ *for some relay* $r'$ *owned by* $RL(r)$ *such that* $r'.sinkRID = senderRID$*.*

When the relay layer of a node $w$ receives a TRANSMIT($m = ((Keys, senderRID, outID), action(parameters))$) message (where $m$ is the message *wrapped* in the TRANSMIT() message), it generally does the following: First, it checks whether $m$ has a valid header for some relay $r'$. In case the header of $m$ reveals that it was to be sent to some existing and alive relay $r$ but did not have a valid header for $r$ anyway, $RL(w)$ replies to the node whose relay layer has RID $senderRID$ with a

NOT-AUTHORIZED() message. In case the message did not have a valid message header and there does not exist such a relay, $RL(w)$ replies with an OUT-RELAY-CLOSED() message. In case of a valid header, if there are invalid keys in $Keys$, $RL(w)$ informs the relay layer that sent the message about these. After this, if $r'$ is not a sink relay, $RL(w)$ then tries to forward the message to the relay that $r'$ has an outgoing connection to. Otherwise, $RL(w)$ tries to deliver the message to $w$. Therefore, it "unpacks" the relay parameters contained in *parameters*: First, it checks whether these relay parameters belong to the same node (otherwise, $m$ is obviously corrupted). Then, for every relay parameter $p$, $RL(w)$ creates a new relay that has an outgoing connection to the relay whose reference was changed to $p$ upon **send**.

While this is the general procedure of $RL(w)$ upon receipt of a TRANSMIT($m = ((Keys, senderRID, outID), action(parameters))$) message, there is an additional detail that needs to be considered here: Recall that when a node $v$ calls SEND($\hat{r}, m$) and $m$ contains references to relays, $RL(v)$ replaces these references by relay parameters containing the necessary information to establish a connection to these relays. Additionally $RL(v)$ inserts $(key, \perp, r)$ into $r'.In$ for every relay $r'$ that was contained in this message. These will be replaced by $(key, RID, \perp)$ after the message has been received by a node. To prevent $(key, \perp, r)$ entries in $.In$ sets for which no corresponding messages in the system exist (which would prevent $.In$ from becoming empty after all other relays have been closed), the relay layers perform a probing to check whether such a message $m$ with a relay parameter with key $key$ exists: It sends a message with a PROBE($ControlKeys, keySequence$) action invocation via $r$ in which $ControlKeys$ is a set containing $key$. If on the path from $r$ to a sink, a relay layer determines that there is a message containing a relay parameter with key $key$ in the buffer of the next relay on the path, the probing for $key$ stops. If this is not the case and the sink does not have a relay with key $key$, a PROBEFAIL() message will be sent in return to inform $r$'s relay layer about this. More specifically, we assume PROBE($ControlKeys, keySequence$) to be a dedicated action type used only for the relay layers. The first parameter is a set and the second parameter is a sequence of keys. To prevent any conflicts with protocols using an action type of the same name, the relay layer could perform a check at the beginning of the SEND() action and rename a conflicting action internally. Since specifying the details of this in the pseudocode would further complicate it, we simply assume that no other protocol uses an action called PROBE. When the first probe message for some specific relay $r$ is sent, $ControlKeys$ contains all keys $key$ such that $(key, \perp, r)$ is contained in some set $r'.In$ for any relay $r'$ owned by the same node. Whenever the probing finds a message with a key $key \in ControlKeys$, it removes $key$ from $ControlKeys$ (or stops further forwarding this probe if $key$ was the last key to be removed from $ControlKeys$). The sequence $keySequence$ is appended at each relay to track the path that the probing has taken. This is required to send back the PROBEFAIL() message to the originator of the probe. Note that the PROBEFAIL() message type contains two parameters: the key that was not found and the sequence of keys that were used to get from the initiator of

the PROBE() message to the sink. This key sequence is used to find the way back to the initiator via the same path (in reverse order) that the probe took.

The full pseudocode of the relay layer upon receipt of a TRANSMIT($m$) is given in Listing 6.2. The action handling the PROBEFAIL() messages is described in pseudocode in Listing 6.3. Note that in the following, we use the variable $RL$ to denote the relay layer that executes this code and the variable $P$ to denote the corresponding node of that relay layer, i.e., $P = w$ such that $RL = RL(w)$.

Listing 6.2: Pseudocode executed by $RL(w)$ when a message $m$ is received by $w$

```
16 TRANSMIT(m = ((Keys, senderRID, outID), action(parameters)))
17   if P owns a relay r' such that r'.state = alive and m has a valid header for r' then
18   | if (key, ⊥, r'') ∈ r'.In for some key ∈ Keys and some relay r'' owned by P such that
19   |  r''.sinkRID = senderRID then
20   | | // first message received via this connection, activate it
21   | | r'.In := r.In \ {(key, ⊥, r'')}
22   | | r'.In := r.In ∪ {(key, senderRID, ⊥)}
23   | let FalseKeys be the set of every key ∈ Keys such that there is no (key, ⊥, r'') ∈ r'.In
24   |  and no (key, senderRID, ⊥) ∈ r'.In
25   | if FalseKeys > 0 then // there were some invalid keys in Keys
26   | | RL.Buf := RL.Buf ∪ {senderRID, NOT-AUTHORIZED(FalseKeys, outID)}
27   | if r'.out.ID = ⊥ then // r' is a sink relay
28   | | if action(parameters) = PROBE(ControlKeys, keySequence) then
29   | | | for every key' ∈ ControlKeys do
30   | | | | if P does not own a relay r'' such that key' ∈ r''.out.Key then
31   | | | | | let (Keys_1, ..., Keys_k) = keySequence
32   | | | | | if there is an RID s. t. (key_k, RID, ⊥) ∈ r'.In for some key_k ∈ Keys_k then
33   | | | | | | let RID be an arbitrary RID such that (key_k, RID, ⊥) ∈ r'.In for some
34   | | | | | |  key_k ∈ Keys_k
35   | | | | | | RL.Buf := RL.Buf ∪ {(RID, PROBEFAIL(key', (Keys_1, ..., Keys_k))}
36   | | else if all IDs of relay parameters of m belong to the same RID then
37   | | | // (otherwise, the message is obviously corrupted)
38   | | | r'.Buf := r'.Buf ∪ {action(parameters)}
39   | | | for each relay parameter (key', ID', level', sRID') in parameters do
40   | | | | if level' > 0 and P does not own a relay r'' such that key' ∈ r''.out.Key then
41   | | | | | create a new relay s with:
42   | | | | | | s.ID := newID, where newID is a new, globally unique ID containing
43   | | | | | |  the RID of RL(w)
44   | | | | | | s.state := alive,
45   | | | | | | s.out := ({key'}, ID'),
46   | | | | | | s.level := level',
47   | | | | | | s.sinkRID := sRID',
48   | | | | | | s.In := ∅, and
49   | | | | | | s.Buf := {TRANSMIT((key', s.ID, ID'), PROBE({}, (key')))}
50   | | | | | replace (key', ID', level', sRID') in parameters by ŝ
51   | | | | else
52   | | | | | replace (key', ID', level', sRID') in parameters by ⊥
53   | else // m needs to be forwarded
54   | | if action(parameters) = PROBE(ControlKeys, keySequence) then
55   | | | append r'.out.Key to keySequence
56   | | | for every key' ∈ ControlKeys do
57   | | | | if there is a message m' ∈ r'.Buf that contains a relay parameter with
58   | | | |  key key' then
```

```
59  |  |  |  |  |  remove key′ from ControlKeys
60  |  |  r′.Buf := r′.Buf ∪ {TRANSMIT(((r′.out.Key, RID(r′.ID), r′.out.ID),
61  |  |    action(parameters)))}
62  else if P owns a relay r′ such that r′.ID = outID and r′.state = alive then
63  |  // m does not have a valid header for r′
64  |  RL.Buf := RL.Buf ∪ {senderRID, NOT-AUTHORIZED(Keys, outID)}
65  else if outID contains the RID of RL(w)
66  |  // P does not own a relay r′ such that r′.ID = outID and r′.state = alive
67  |  RL.Buf := RL.Buf ∪ {senderRID, OUT-RELAY-CLOSED(outID))}
```

Listing 6.3: Pseudocode executed upon PROBEFAIL($key, keySequence$)

```
68  PROBEFAIL(key, keySequence)
69  let (Keys_1, ..., Keys_k) = keySequence // k = |keySequence|
70  if P owns a relay r such that Keys_k ∩ r.out.Key ≠ ∅ then
71  |  if k > 1 then // needs to be passed on
72  |  |  if there is an RID such that (key_{k−1}, RID, ⊥) ∈ r.In for some key_{k−1} ∈ Keys_{k−1} then
73  |  |  |  let RID be an arbitrary RID s. t. (key_{k−1}, RID, ⊥) ∈ r.In for some key_{k−1} ∈ Keys_{k−1}
74  |  |  |  RL.Buf := RL.Buf ∪ {(RID, PROBEFAIL(key, (Keys_1, ..., Keys_{k−1})))}
75  |  else
76  |  |  if P owns a relay r′ such that (key, ⊥, r) ∈ r′.In then
77  |  |  |  r′.In := r′.In \ (key, ⊥, r)
```

When a NOT-AUTHORIZED($Keys, outID$) control message is received, the relay layer considers each $key \in Keys$ individually: if there is a non-sink relay $r$ with $r.out.ID = outID$ and $key \in r.out.Key$, the relay layer removes $key$ from $r.out.Key$. If the result of this is that there is no key left in $r.out.Key$, all elements $(key, \perp, r)$ are removed from $r'.In$ for every relay $r'$ and $r$ is deleted. This is because without any key in $r.out.Key$, $r$ would not be able to forward any message received by $r$ anyway and thus $r$ should consequently be deleted. The pseudocode of the NOT-AUTHORIZED() action is given in Listing 6.4.

Listing 6.4: Pseudocode executed upon NOT-AUTHORIZED($m$)

```
78  NOT-AUTHORIZED(Keys, outID)
79  for every key ∈ Keys do
80  |  if there exists a relay r with r.out.ID = outID ≠ ⊥ and key ∈ r.out.Key then
81  |  |  r.out.Key := r.out.Key \ {key}
82  |  |  if r.out.Key = 0 then // outgoing link of r is broken / closed
83  |  |  |  // remove all "pending" (unconfirmed) relays sent via r
84  |  |  |  for all relays r′ do
85  |  |  |  |  for all keys key′ such that (key′, ⊥, r) ∈ r′.In do
86  |  |  |  |  |  r′.In := r′.In \ {(key′, ⊥, r)}
87  |  |  |  delete r̂
88  |  |  |  completely remove r
```

The TIMEOUT action mainly detects and corrects all values that are obviously corrupted and contradict the definition of a legal state that will be given later. In addition, for each relay $r$ it serves the following purposes: First, it periodically sends a PING($r.ID, r.level, r.sinkRID, key$) message to every relay layer whose RID is contained as the second parameter of a triple $(key, RID, \perp)$ in $r.In$. This is to give connected relays $r'$ with $r'.out.ID = ID$ and $key \in r'.out.Key$ the opportunity to

correct their level or sinkRID information and also to determine if there are relays in $r.In$ that do not exist. Second, the TIMEOUT action detects and completely removes deleted relays that do not need to be kept anymore (e.g., because all of their messages have been transmitted) and it also shuts down the relay layer if the node is inactive and all relays of it have been removed completely. When a dead relay $r$ that is not a sink is removed completely, the action additionally sends out an IN-RELAY-CLOSED$(r.out.Key, RID(r), r.out.ID)$ message as to inform the relay layer of the relay with ID $r.out.ID$ that $r$ has been closed. Third, its sends out the aforementioned PROBE() messages. The full pseudocode of this action is given in Listing 6.5.

Listing 6.5: Pseudocode of the periodically executed TIMEOUT action

```
89  TIMEOUT
90    for all relays r owned by P do
91    | if r.ID does not contain the RID of P or there is another relay r' s. t. r'.ID = r.ID then
92    | | delete r̂
93    | | for all relays r' do
94    | | | for all keys key such that (key, ⊥, r) ∈ r'.In do
95    | | | | r'.In := r'.In \ {(key, ⊥, r)}
96    | | completely remove r
97    | if r.out.ID = ⊥ then
98    | | r.level = 0
99    | else if r.level < 1 then
100   | | r.level := 1
101   | if r.out is not a pair (Key, ID) such that (i) Key is a set and (ii) ID = ⊥ or ID contains
102   |   the RID of some relay layer then
103   | | delete r̂
104   | | for all relays r' do
105   | | | for all keys key such that (key, ⊥, r) ∈ r'.In do
106   | | | | r'.In := r'.In \ {(key, ⊥, r)}
107   | | completely remove r
108   | if r.out.ID = ⊥ and r.out.Key ≠ ∅ then
109   | | r.out.Key := ∅
110   | if r.out.ID ≠ ⊥ and r.out.Key = ∅ then
111   | | delete r̂
112   | | for all relays r' do
113   | | | for all keys key such that (key, ⊥, r) ∈ r'.In do
114   | | | | r'.In := r'.In \ {(key, ⊥, r)}
115   | | completely remove r
116   | for all (key, X, Y) ∈ r.In do
117   | | if there is a (key, X', Y') ∈ r.In such that X' ≠ X and Y' ≠ Y
118   | |   or there is a (key, X', Y') ∈ r'.In for some relay r' ≠ r
119   | |   or key does not belong to RL then
120   | | | r.In := r.In \ {(key, X, Y)}
121   | for all (key, RID, ⊥) ∈ r.In do
122   | | RL.Buf := RL.Buf ∪ {(RID, PING(r.ID, r.level, r.sinkRID, key))}
123   | for all x ∈ r.In such that x ≠ (key, RID, ⊥) for any RID RID and x ≠ (key, ⊥, r') for any
124   |   existing relay r' such that r'.out.ID ≠ ⊥ do
125   | | r.In := r.In \ {x}
126   | if r.state = dead then
127   | | r.In := ∅
```

```
128     |  |  if r.out.ID = ⊥ then
129     |  |  |  completely remove r
130     |  |  else if r.Buf = ∅ and P does not own a relay r′ such that (key, ⊥, r) ∈ r′.In for some
131     |  |    key key then
132     |  |  |  |  RL.Buf := RL.Buf ∪ {(RID(r.out.ID), In-relay-closed(r.out.Key, RID(P),
133     |  |  |  |    r.out.ID))}
134     |  |  |  completely remove r
135     |  if P is inactive and (i) r.out.ID = ⊥ or (ii) r.In = ∅ and r.Buf = ∅ and there is no
136     |    relay r′ such that (key, ⊥, r) ∈ r′.In for some key key then
137     |  |  completely remove r
138     |  if P owns a relay r′ ≠ r s. t. there is a key ∈ r′.out.Key s. t. key ∈ r.out.Key then
139     |  |  if r′.ID > r.ID then
140     |  |  |  delete r̂
141     |  |  |  for all relays r″ do
142     |  |  |  |  for all keys key such that (key, ⊥, r) ∈ r″.In do
143     |  |  |  |  |  r″.In := r″.In \ {(key, ⊥, r)}
144     |  |  |  completely remove r
145     |  let keySequence be a sequence consisting of the single element r.out.Key
146     |  let ControlKeys be the set of all keys key′ s. t. P owns a relay r′ s. t. (key′, ⊥, r) ∈ r′.In
147     |    and there is no message in r.Buf containing a relay parameter with key key′
148     |  if P is active or r.In ≠ ∅ or ControlKeys ≠ ∅ then
149     |  |  r.Buf := r.Buf ∪ {Transmit((r.out.Key, RID(r.ID), r.out.ID), Probe(ControlKeys,
150     |  |    keySequence))}
151     if P is inactive and P owns no relay then
152     |  shut down this relay layer completely
```

When a relay layer receives a $\textsc{Ping}(ID, level, sinkRID, key)$ message it checks whether there is a corresponding relay $r$ with $r.out.ID = ID$ and $key \in r.out.Key$. If there is no such relay, it responds to the relay layer owning the relay with ID $ID$ with an In-relay-closed() message indicating that there is no such relay with such a key. Otherwise, if $r.level > level + 1$, it updates $r.level$ to $level + 1$ and $r.sinkRID$ to $sinkRID$. If $r.level < level + 1$, it deletes $r$ (in this case correcting the value would be dangerous as this would allow for cycles in the relay graph). The pseudocode of the Ping() message is given in Listing 6.6 and the pseudocode of the $\textsc{In-relay-closed}(Keys, senderRID, ID)$, which basically removes every entry $(key, RID, \bot)$ from all $.In$ sets such that $key \in Keys$, is given in Listing 6.7.

Listing 6.6: Pseudocode of the Ping() action

```
153 Ping(ID, level, sinkRID, key)
154   if ID ≠ ⊥ then
155   |  if P owns a relay r such that r.out.ID = ID and key ∈ r.out.Key do
156   |  |  r.sinkRID := sinkRID
157   |  |  if r.level > level + 1 then
158   |  |  |  r.level := level + 1
159   |  |  if r.level < level + 1 then
160   |  |  |  delete r̂
161   |  |  |  // remove all "pending" (unconfirmed) relays sent via r
162   |  |  |  for all relays r′ do
163   |  |  |  |  for all keys key such that (key, ⊥, r) ∈ r′.In do
164   |  |  |  |  |  r′.In := r′.In \ {(key, ⊥, r)}
165   |  |  |  completely remove r
```

```
166     | else
167     | | RL.Buf := RL.Buf ∪ {(RID(ID), In-relay-closed({key}, RID(P), ID))}
```

Listing 6.7: Pseudocode of the In-relay-closed() action

```
168 In-relay-closed(Keys, senderRID, ID)
169     for every key ∈ Key do
170     | if P owns a relay r such that r.ID = ID and (key, senderRID, ⊥) ∈ r.In then
171     | | r.In := r.In \ {(key, senderRID, ⊥)}
```

When **delete** $\hat{r}$ is called, $RL(r)$ sets $r.state$ to *dead* and sends an Out-relay-closed($r.ID$) message to every relay layer whose RID is the second parameter of a triple in $r.In$. Afterwards, it empties $r.In$ so that no message can be received via $r$ from that point in time. The pseudocode of this is given in Listing 6.8.

Listing 6.8: Pseudocode executed upon **delete** $\hat{r}$

```
172 delete r̂
173     r.state := dead
174     for every (key, RID, ⊥) ∈ r.In do
175     | RL.Buf := RL.Buf ∪ {(RID, Out-relay-closed(r.ID))}
176     r.In := ∅
177     if r.out.ID = ⊥
178     | completely remove r
```

Note that a non-sink relay $r$ is not closed immediately during the execution of **delete** $\hat{r}$. This is for two reasons: First, the complete removal of $r$ is postponed to allow all messages still in $r.Buf$ to be delivered first. Second, there might still be relays $r'$ such that $(key, \bot, r) \in r'.In$ for some key $key$: i.e., a reference of $r'$ was sent via $r$ and the key created thereby is still unconfirmed. Would $r$ be removed immediately, the probing for these entries could not be executed anymore and $(key, \bot, r)$ would be erroneously removed from $r'.In$ even if the message carrying the corresponding relay parameter will be delivered later on. As soon as $r.Buf$ is empty and all relay keys sent via $r$ have been confirmed or deleted (i.e., there is no longer any relay $r'$ such that $(key, \bot, r) \in r'.In$), $r$ will be removed completely upon the execution of Timeout.

When a relay layer receives an Out-relay-closed($ID$) message and owns a non-sink relay $r$ with $r.out.ID = ID$, it removes all triples $(key, \bot, r)$ from $r'.In$ for every relay $r'$ owned by it, calls **delete** $\hat{r}$ and completely removes $r$ afterwards. The pseudocode of this action can be found in Listing 6.9.

Listing 6.9: Pseudocode of the Out-relay-closed() action

```
179 Out-relay-closed(ID)
180     if ID ≠ ⊥ and P owns a relay r such that r.out.ID = ID then
181     | // remove all "pending" (unconfirmed) relays sent via r
182     | for all relays r' do
183     | | for all keys key such that (key, ⊥, r) ∈ r'.In do
184     | | | r'.In := r'.In \ {(key, ⊥, r)}
185     | delete r̂
186     | completely remove r
```

### 6.2.5. Additional Terms and Definitions

In order to define legal states for the relay layer, we introduce the notion of a *valid relay*. For this definition, for a node $u$, we say a message $m$ is *in transit to* $RL(u)$ if and only if there is a relay layer $R$ in the system such that $(RID(RL(u)), m) \in R.Buf$ or there is a relay $r'$ owned by $R$ such that $m \in r'.Buf$ and $RID(r'.out.ID) = RID(u)$.

**Definition 6.3** (Valid Relay)**.** *A relay $r$ is* valid *if and only if*

1. *$r.state = alive$ and there is no* Out-relay-closed$(r.ID)$ *message in the system,*
2. *$r.ID$ is globally unique and contains the* RID *of $RL(r)$,*
3. *$r.out$ stores a pair (Key, ID) such that $Key$ is a set, and*
4. *for every $key \in r.out.Key$, there is no relay $r''' \neq r$ owned by the same node such that $key \in r'''.out.Key$, and* ***either***
5. *$r$ is a sink, i.e., $r.out = (\{\}, \perp)$, $r.level = 0$, $r.sinkRID = RID(r)$ and the node owning $r$ is active* ***or***
6. *a) $r.out.ID \neq \perp$ and there is a valid relay $r'$ with $r'.ID = r.out.ID$,*
   *b) $r.level = r'.level + 1 > 0$ and $r.sinkRID = r'.sinkRID$, and*
   *c) there is at least one key feasible for $r$ (see below) in $r.out.Key$, and*
7. *$r.In$ only consists of triples $(key, RID, \perp)$ with $RID \neq \perp$ or $(key, \perp, r'')$ for a relay $r''$ owned by $RL(r)$ such that $r''.out.ID \neq \perp$,*
8. *every key key used as a first parameter of a triple in $r.In$ is locally unique (i.e., it does not appear in any other triple in $r.In$ or $r''.In$ for any relay $r'' \neq r$) and belongs to the node owning $r$,*
9. *there is no* Ping$(r.ID, level, sinkRID, key)$ *message in the system such that $level \neq r.level$ or $sinkRID \neq r.sinkRID$ or $(key, \perp, r'') \in r.In$ for any relay $r''$, and*
10. *for every $(key, RID, \perp) \in r.In$ there is no* Not-authorized$(Keys, r.ID)$ *message in transit to the relay layer with* RID *$RID$ such that $key \in Keys$, and for every $(key, \perp, r'') \in r.In$ there is no* Not-authorized$(Keys, r.ID)$ *message in transit to the relay layer with* RID *$r''.senderRID$ such that $key \in Keys$.*

*Note that for a valid relay,* ***all*** *of Properties 1–4 and Properties 7–10 and* ***one*** *of Properties 5 and 6 need to hold.*

The above definition uses the notion of a *feasible key* defined as follows:

**Definition 6.4** (Feasible key)**.** *A key key is* feasible *for a relay $r$ if and only if $key \in r.out.Key$ and there is a relay $r'$ such that $r.out.ID = r'.ID$ and there is no* In-relay-closed$(Keys, RID(r), r.out.ID)$ *message in transit to $RL(r')$ such that $key \in Keys$, and*

*(i)* $(key, RID(r), \bot) \in r'.In$ *or*

*(ii)* $(key, \bot, r'') \in r'.In$ *for an out-confirmed (see below) relay $r''$ owned by $RL(r')$ such that $r''.sinkRID = RID(r)$ and there is no* Probefail$(key, keySequence)$ *message in the system and for the sequence of relays $(r_1, r_2, \ldots, r_k)$ such that $r_1 = r''$, $r_{i+1}.ID = r_i.out.ID$ for all $1 \leq i < k$ and $r_k.out.ID = \bot$, there is no* Probe$(ControlKeys, keySequence)$ *message such that $key \in ControlKeys$ in $r'''.Buf$ for any relay $r''' \notin \{r_1, \ldots, r_{k-1}\}$.*

The above definition uses the notion of an *out-confirmed* relay. We now introduce this term formally. Before this, we have to define a *confirmed relay*, though:

**Definition 6.5** (Confirmed Relay)**.** *A relay $r$ is* confirmed *if and only if*

*(i) $r$ is valid and*

*(ii) if $r.out.ID \neq \bot$ then there is a confirmed relay $r'$ such that $r.out.ID = r'.ID$ and there is a key key feasible for $r$ such that $(key, RID(r), \bot) \in r'.In$.*

Equipped with this definition, we define an out-confirmed relay as follows:

**Definition 6.6** (Out-confirmed)**.** *A relay $r$ is* out-confirmed *if and only if*

*(i) Properties 2–4 and Property 6 are satisfied for $r$,*

*(ii) the relay $r'$ such that $r.out.ID = r'.ID$ is confirmed, and*

*(iii) there is a key key feasible for $r$ such that $(key, RID(r), \bot) \in r'.In$.*

Note that whenever Definition 6.5 or Definition 6.6 require a key to be feasible, this key must be feasible according to the first alternative of Definition 6.4. Therefore, the notion of a feasible key is well-defined.

For the proofs of correctness, some additional notions will turn out to be helpful. We start by defining a relay as *lingering* if certain properties are fulfilled:

**Definition 6.7** (Lingering relay)**.** *A relay $r$ is called* lingering *if and only if $r.out.ID \neq \bot$ and:*

*(i) $r.state = alive$,*

*(ii) there is a relay $r'$ owned by $RL(r)$ such that $(key, \bot, r) \in r'.In$ for some key, or*

*(iii) $r.Buf \neq \emptyset$*

For an intuition behind this definition, observe that lingering relays are not removed completely when they are deleted. This is postponed to until the relay is no longer lingering, in which case we say the relay is *released*:

**Definition 6.8** (Releasing a lingering relay)**.** *A lingering relay $r$ is* released *as soon as for the first time, $r.state = dead$, there is no relay $r'$ owned by $RL(r)$ such that $(key, \perp, r) \in r'.In$ for any key and $r.Buf = \emptyset$.*

Note that it is not possible to send new messages via lingering relays $r$ such that $r.state = dead$. However, under certain conditions, messages still in $r.Buf$ can be delivered. Moreover, for the relays $r'$ such that $(key, \perp, r) \in r'.In$, $r$ still serves the purpose for $r'$ described in Section 6.2.4. To formalize these conditions, we introduce the following definition:

**Definition 6.9** (Lingering-valid relay)**.** *A relay $r$ is called* lingering-valid *if and only if $r$ is lingering and all but Property 1 and Property 5 of a valid relay hold for $r$.*

Note that Property 1 may hold for a lingering-valid relay, but is not required to do so. In particular, every valid relay $r$ such that $r.out.ID \neq \perp$ is lingering-valid as well.

We also introduce the following, quite similar definition that will turn out to be helpful for the definition of a legal state:

**Definition 6.10** (Dead-valid relay)**.** *A relay $r$ is called* dead-valid *if and only if $r.state = dead$ and all but Property 1 and Property 5 of a valid relay hold for $r$.*

Observe in Definition 6.9 and Definition 6.10 that $r.out.ID \neq \perp$ for a lingering-valid or dead-valid relay $r$.

We also specify some properties that need to be fulfilled for *valid relay parameters* as follows:

**Definition 6.11** (Valid relay parameter)**.** *A relay parameter $(key, ID, level, sinkRID)$ contained in a message $m$ in a buffer $r.Buf$ of a relay $r$ is valid if and only if*

1. *$r$ is lingering-valid,*
2. *there is no other relay parameter with key key in the system,*
3. *$ID \neq \perp$ and there is a valid relay $r'$ with $r'.ID = ID$ in the system,*
4. *$level = r'.level + 1$ and $sinkRID = r'.sinkRID$,*
5. *$(key, \perp, r'') \in r'.In$ for some lingering-valid relay $r''$ owned by the same node as $r'$ such that $r''.sinkRID = r.sinkRID$ and some key key that belongs to $RL(r')$,*
6. *all $ID$s contained in relay references in $m$ belong to the same* RID*,*

7. *there is no relay* $r'''$ *in the system such that* $key \in r'''.out.Key$,

8. *there is no message* $m$ *in the system using key* $key$,

9. *there is no* PROBEFAIL$(key, keySequence)$ *message in the system for any* $keySequence$,

10. *for every* PROBE$(ControlKeys, keySequence)$ *message* $m'$ *in the system such that* $key \in ControlKeys$, *there is a sequence of relays* $(r_1 = r'', r_2, \ldots, r_k, \ldots, r_s)$, $s \geq k+2$ *such that* $r_{i+1}.ID = r_i.out.ID$ *for all* $i \in \{1, \ldots, s-1\}$, $m'$ *is stored in* $r_k.Buf$, *and* $r = r_j$ *for some* $j \in \{k+1, \ldots, s-1\}$,

11. *there is no* IN-RELAY-CLOSED$(Keys, RID, ID)$ *message in the system with* $key \in Keys$,

12. *the header of* $m$ *is* $(Keys, RID(r.ID), r.out.ID)$ *such that there is a* $key' \in Keys$ *that is feasible for* $r$, *and*

13. *there is a sequence of relays* $(r_1 = r'', r_2, \ldots, r_k)$ *such that* $r_{i+1}.ID = r_i.out.ID$ *for all* $1 \leq i < k$, $r_k = r$, *and for every* $r_j$ *such that* $1 \leq j < k$ *there is a* $key \in r_j.out.Key$ *such that* $(key, RID(r_j), \perp) \in r_{j+1}.In$ *and there is no* IN-RELAY-CLOSED$(Keys, RID(r_j), r_j.out.ID)$ *message in transit to* $RL(r_{j+1})$ *such that* $key \in Keys$.

Using the previous definitions, we can define a *valid relay graph* as follows:

**Definition 6.12** (Valid relay graph)**.** *A* valid relay graph *of a system state* $S$ *is the subgraph of the relay graph* $G = (R \cup P, E_P \cup E_{Ch})$ *of* $S$ *such that every* $r \in R$ *such that* $r.state = alive$ *is valid, every* $r \in R$ *such that* $r.state = dead$ *is dead-valid, and every* $(v, w) \in E_{Ch}$ *is due to a valid relay parameter.*

Given the definition of a valid relay graph, we can define a legal state as follows:

**Definition 6.13** (Legal state)**.** *A state* $S$ *is* legal *if and only if there is no difference between the relay graph of* $S$ *and its valid relay graph.*

Note that Property 6b) of a valid relay and the fact that $r.level \in \mathbb{N}_0$ for every relay $r$ implies the following:

**Corollary 6.14.** *Every valid relay graph is cycle-free.*

The definition of a valid relay also implies the following:

**Corollary 6.15.** *For every valid non-sink relay* $r_1$, *there is a unique relay* $r_k$ *such that* $r_k.out.ID = \perp$ *and there exist* $r_2, \ldots, r_{k-1}$ *such that* $r_{i+1}.ID = r_i.out.ID$ *for all* $1 \leq i < k$. $r_k$ *is called the* sink *of* $r$.

As we will show later on, this leads to that the node owning the sink of a relay $r$ will be the one that receives messages sent via $r$. This makes it reasonable to give it a dedicated name:

**Definition 6.16** (Sink node)**.** *For a valid non-sink relay $r$, the node owning the sink of $r$ is called the* sink node *of $r$. For a valid sink relay $r$, the node owning $r$ is called the* sink node *of $r$.*

In general, an application can easily destroy connectivity by deleting relays with incoming connections (i.e., relays $r$ such that $r.In \neq \emptyset$). Since this is usually not desired, we define an application to be *deliberate* if it does not do so. More formally:

**Definition 6.17** (Deliberate application)**.** *An application is* deliberate *if and only if it does not delete a relay $r'$ such that $r'.In \neq \emptyset$.*

Note that the definition of a deliberate application includes that the application does not call **stop** as long as there are sink relays with incoming connections.

Observe that when a relay $r$ ceases to exist, this can be for two reasons: One is that it had been dead and is now ready to be removed completely. The other is that it is merged (with other relays) into a new relay $r'$. To formalize the semantic relationship between $r$ and $r'$ in this case, we say that $r'$ is a *successor* of $r$. Since $r'$ itself may be merged again, $r$ may have multiple successors in fact. To conceptualize this and to work with this in the analysis, we introduce the following definition:

**Definition 6.18** (Successor)**.** *For two relays $r, r'$, $r'$ is called a* successor *of $r$ if and only if $r'$ is the relay resulting from a call of **merge**$(R)$ such that $r \in R$ or $r'$ is the relay resulting from a call of **merge**$(R)$ such that $r'' \in R$ and $r''$ is a successor of $r$. We denote by $s(r)$ the set of all successors of $r$ including $r$.*

### 6.2.6. Main Results

We now state the main results concerning the relay layer.

**Theorem 6.19.** *Every dead relay is eventually removed completely.*

This result is important because relays are not always removed immediately after **delete** has been executed on them. For example, when some other relay reference has been sent via a relay and not yet received by the sink, the relay continues to persist.

**Theorem 6.20.** *If for every indirect relay $r$ the application eventually deletes a successor of $r$, all relay layers of inactive nodes will eventually be shut down.*

This result is worth mentioning because the relay layer of a node that issues **stop** is not always shut down immediately. For example, when a node has a non-sink relay with incoming connections, the relay layer does not shut down before the incoming connections have been closed.

**Theorem 6.21.** *If the application is deliberate, every message sent via a valid relay $r$ will be received by the sink node of $r$.*

This result proves that the node owning the sink of a relay $r$ receives all messages sent via $r$.

**Theorem 6.22.** *If the application is deliberate, in every computation that starts in a legal state, every state is legal.*

In other words, the result of Theorem 6.22 resembles the convergence property of self-stabilization.

**Theorem 6.23.** *If the application is deliberate and does not send any reference via a relay that is not valid and for some arbitrary but fixed $l \in \mathbb{N}$ does not send the reference of a relay $r$ such that $r.level \geq l$, every computation will reach a legal state.*

The result of Theorem 6.23 basically resembles the closure property under certain reasonable conditions. Putting Theorem 6.22 and Theorem 6.23 together, we obtain the following corollary:

**Corollary 6.24.** *Every computation such that the application is deliberate and does not send any reference via a relay that is not valid and for some arbitrary but fixed $l \in \mathbb{N}$ does not send the reference of a relay $r$ such that $r.level \geq l$ contains a suffix of legal states.*

This result states that the relay layer is self-stabilizing under certain reasonable conditions. Corollary 6.24 also immediately implies the following:

**Corollary 6.25.** *If the application does not issue any commands, starting from any initial state $S$ the system will reach a state $S'$ such that $S'$ and every subsequent state are legal.*

This result states the same as Corollary 6.24 though under stronger conditions. It is worthwhile to consider this claim on its own because it resembles the classical definition of self-stabilization in which it is assumed that, starting from the initial state, no change occurs to the system other than by the self-stabilizing protocol.

**Theorem 6.26.** *In every computation consisting only of legal states, the following holds for every alive relay $w$: Whenever* ***has-incoming($\hat{w}$)*** *returns false, then there is no relay $v$ with an edge $(v, w)$ in the relay graph. Moreover, if the computation has a suffix in which there is no relay $v$ that has an edge $(v, w)$ to $w$ in the relay graph and $w$ is not deleted as long as* ***has-incoming($\hat{w}$)*** *is true, there is a state $S$ such that* ***has-incoming($\hat{w}$)*** *returns false in $S$ and every subsequent state until $w$ is deleted.*

This result states that **has-incoming** resembles a kind of a "delayed" $\mathcal{NIDEC}$ oracle (which, as we will see, is sufficient to be able to solve the $\mathcal{FDP}$): If **has-incoming($\hat{r}$)** returns false, there is no incoming connection to $r$ and no message could be delivered to $r$ anymore, which resembles that $\mathcal{NIDEC}$ is true. On the other hand, if there is no incoming connection to $r$, which includes that no message can be delivered to $r$ anymore (note that if there was some message to be delivered to $r$ that has not been delivered yet, it would need to be in the buffer of some relay $r'$ sharing an edge with $r$ in the relay graph and $r'$ would be valid or dead-valid because we assume the computation to consist only of legal states), i.e., $\mathcal{NIDEC}$ is and remains true, then eventually **has-incoming($\hat{r}$)** will return false (unless $r$ is deleted before).

## 6.3. Self-Stabilization Proofs

In this section we present the proofs of the theorems mentioned in Section 6.2.5. Their proofs rely on several lemmas that will be proven throughout this section. Since the number of lemmas required for all the theorems is quite large, the section is divided into several subsections. We begin in Section 6.3.1 with some preliminaries. After that, we provide the proofs necessary for Theorem 6.19 and Theorem 6.20 in Section 6.3.2. The proof of Theorem 6.22 relies on some of these results and some additional lemmas that are proven in Section 6.3.3. For the proof of Theorem 6.22, some additional results are required whose proofs are provided in Section 6.3.4. Then, in Section 6.3.5, we present the missing proofs for Theorem 6.23, Corollary 6.24 and Corollary 6.25. Last, in Section 6.3.6, we provide the proof of Theorem 6.26.

### 6.3.1. Preliminaries

Here we make some additional definitions that are used for the analysis only.

First of all, building on the definition of a successor of a relay (Definition 6.18), we define the *current successor* of a relay in a particular state:

**Definition 6.27** (Current successor)**.** *For a relay $r$ and a state $S$, if there is $r' \in s(r)$ that exists during $S$, $r'$ is the* current successor *of $r$, denoted by $cs_S(r)$. If there is no $r' \in s(r)$ that exists during $S$, $cs_S(r) = \times$. Note that if the state in question is clear from the context, we may omit $S$ and refer to the current successor of $r$ by $cs(r)$.*

As it turns out, it will be very convenient to be able to refer to the *last successor* $ls(r)$ of a relay $r$ defined as follows:

**Definition 6.28** (Last successor)**.** *For a relay $r$, the* last successor *of $r$, $ls(r)$ is defined as follows: If $s(r)$ is finite and the relay $r' \in s(r)$ such that $s(r') = \{r'\}$ (i.e., it does not have a successor) is never removed completely, then $ls(r) = r'$. If $s(r)$ is finite and the relay $r' \in s(r)$ such that $s(r') = \{r'\}$ is removed completely at some point in time, then $ls(r) = \times$. Otherwise (if $s(r)$ is infinite), $ls(r) = \perp$.*

One can observe that PROBE() messages are regularly sent out during TIMEOUT under certain conditions. Intuitively these conditions are that either the corresponding relay is alive or that something prevents that relay from being removed completely even though the relay is dead. Since we will often refer to relays that fulfill these specific conditions, we formally define a relay to be *probing* when the following conditions are met:

**Definition 6.29** (Probing relay)**.** *A relay is* probing *if and only if:*

*(i) the node owning $r$ is active,*

*(ii) $r.In \neq \emptyset$, or*

*(iii) there is a relay $r'$ owned by $RL(r)$ such that $(key, \perp, r) \in r'.In$ for some key $key$.*

Note that in the following, unless stated differently whenever we refer to a numbered property for a relay, we mean the property of Definition 6.3. Likewise, whenever we refer to a numbered property for a relay parameter, we mean the property of Definition 6.11. Furthermore, when we refer to line numbers, we refer to the pseudocode given in Listings 6.1 to 6.9.

### 6.3.2. Proofs for Theorem 6.19 and Theorem 6.20

In order to prove Theorem 6.19 and Theorem 6.20, we proceed as follows: First, we prove that for every pair of relays $r, r'$ such that $r'.ID = r.out.ID$, there is eventually a confirmed key feasible for $r$ in $r'.In$ or $r$ is removed completely. This implies that in case $r$ continues to exist, $RL(r')$ will eventually send corresponding PING() messages to $RL(r)$. We exploit this to show that there are no infinite cycles in relay connections. In fact, we can show that every relay that is not removed completely in finite time has a sink, which it is connected to via a sequence of alive relays. Putting these results together, we obtain that PROBE() messages will successfully reach their targets. This is important to show that every dead relay is removed completely: The only reason for why a dead relay could be prevented from being removed completely is (besides the fact that its buffer is not yet empty, which is a matter of time) that it is a relay via which one or more yet unconfirmed keys were sent. The fact that the probing can be done correctly ensures that these keys will eventually become confirmed and the dead relay is removed completely.

After we have proven Theorem 6.19 this way, there is only one additional aspect that needs to be considered for Theorem 6.20: Relays of inactive nodes are not deleted immediately because the relay layer waits for their $In$ set to become empty first. This can be proven to always happen if the application does not keep any indirect relay or its successors alive forever. Equipped with this proof, the claim of the theorem follows immediately.

**Lemma 6.30.** *For every alive relay $r$ such that there is a relay $r'$ such that $r'.ID = r.out.ID$, the following holds: if $ls(r) \neq \times$, then $r'$ is and remains alive*

*(and is not merged) and there is a state $S$ such that there is a feasible key $key$ in $cs_{S'}(r).out.Key$ in every state $S' \geq S$ and $(key, RID(r), \perp) \in r'.In$ in every state $S' \geq S$.*

*Proof.* Consider an arbitrary alive relay $r$ such that $r.out.ID \neq \perp$ and there is a relay $r'$ such that $r'.ID = r.out.ID$. Furthermore assume that $ls(r) \neq \times$.

First of all, note that no $key$ is ever added to $r_s.out.Key$ for any $r_s \in s(r)$, because this only happens during the initial creation of a relay (i.e., when a relay is created due to the receipt of a relay parameter). We now prove that in some state $S_1$, for every $key \in r.out.Key$ either there is no IN-RELAY-CLOSED$(Keys, RID(r), r.out.ID)$ message with $key \in Keys$ or $key \notin cs_{S_1}(r).out.Key$.

Consider an arbitrary $key \in r.out.Key$ and assume that $key \in cs(r).out.Key$ forever (otherwise, we are done). Note that according to the pseudocode, only $RL(r)$ could send an IN-RELAY-CLOSED$(Keys, RID(r), r.out.ID)$ message with $key \in Keys$. However, according to the pseudocode, $RL(r)$ only does so if it completely removes $cs(r)$ immediately thereafter (Line 132) or when $cs(r)$ does not exist (Line 167), which contradicts $ls(r) \neq \times$, or if $RL(r)$ receives a PING$(r.out.ID, level, sinkRID, key)$ message and $key \notin r.out.Key$, which contradicts $key \in Keys$. Thus as soon as all IN-RELAY-CLOSED() messages initially in the system have been transmitted and processed, no IN-RELAY-CLOSED$(Keys, RID(r), r.out.ID)$ message with $key \in Keys$ will be created.

What we have proven so far is that at state $S_1$, for every $key \in r.out.Key$, there is no IN-RELAY-CLOSED$(Keys, RID(r), r.out.ID)$ message with $key \in Keys$ and there will never be such a message. Note here and in the following that $cs(r).out.Key$ can never become empty, because in this case, $cs(r)$ would be removed completely in Line 88 or Line 115, yielding a contradiction to $ls(r) \neq \times$.

We now show that $r'$ is and remains alive and that for every $key \in cs(r).out.Key$, eventually there will be a triple $(key, RID(r), \perp) \in r'.In$ or $key \notin cs(r).out.Key$. Note that in every state, the node owning $r$ must be alive or $cs(r).In \neq \emptyset$ or there is a relay $r''$ such that $(key, \perp, cs(r)) \in r''.In$ because otherwise $cs(r).Buf$ would become empty and $cs(r)$ would be removed completely in Line 137 upon TIMEOUT. Thus, during the next execution of TIMEOUT, $RL(r)$ will insert a message TRANSMIT$((cs(r).out.Key, RID(r.ID), r.out.ID), $PROBE$(\{ControlKeys\}, (cs(r).out.Key))))$ into $cs(r).Buf$ (see Line 149). Consider the action executed by $RL(r')$ upon receipt of such a message TRANSMIT$(((Keys, RID(r.ID), r.out.ID),$ PROBE$(\{ControlKeys\}, Keys)))$: $RL(r')$ first checks whether this message has a valid header for $r'$ (see Line 17). If this is the case, then by Definition 6.2 $(key, RID(r), \perp) \in r'.In$ (in which case we are done) or $(key, \perp, r'') \in r.In$ for some relay $r''$ owned by $RL(r')$ such that $r''.sinkRID = RID(r)$. In the latter case, $RL(r')$ will replace $(key, \perp, r'')$ by $(key, RID(r), \perp)$, so we are done in this case as well (see Lines 21–22). Thus assume that the message does not have a valid header for $r'$. If $r'.state = dead$ or $r'$ does not exist, $RL(r')$ will respond with an OUT-RELAY-CLOSED$(r.out.ID)$ message (see Line 67) causing $cs(r)$ to be removed completely (see Line 186), which represents a contradiction. This also proves that

$r'$ is and remains alive. Otherwise, Line 64 will be executed in this case, having the effect that $RL(r')$ will send a NOT-AUTHORIZED$((Keys, r.out.ID)$ message to $RL(r)$. Upon receipt of this message, $RL(r)$ will remove every $key \in Keys$ from $cs(r).out.Key$ (see Line 81). All in all, we have now proven that for every $key \in cs(r).out.Key$, eventually there will be a triple $(key, RID(r), \perp) \in r'.In$ or $key \notin cs(r).out.Key$. Since $cs(r).out.Key$ cannot become empty as argued before and because there cannot be an IN-RELAY-CLOSED$(Keys, RID(r), r.out.ID)$ message with $key \in Keys$ for any $key \in cs(r).out.Key$ anymore, the claim of the lemma follows. □

**Lemma 6.31.** *In every cycle of relays $(r_1, r_2, \ldots, r_k)$ such that $r_i.out.ID = r_{i+1}.ID$ for every $i \in \{1, 2, \ldots, k-1\}$ and $r_k.out.ID = r_1$, there is a relay $r_l$ such that $r_l$ will be removed completely.*

*Proof.* Consider an arbitrary cycle of relays $(r_1, r_2, \ldots, r_k)$ such that $r_i.out.ID = r_{i+1}.ID$ for every $i \in \{1, 2, \ldots, k-1\}$ and $r_k.out.ID = r_1$. Assume for contradiction that none of the relays in this cycle will be removed completely in finite time (since a relay is also removed completely in the case of a merge, this implies $cs_S(r_i) = r_i$ for every $r_i$ in every state $S$). Lemma 6.30 then implies that eventually for every relay $r_i$ in the cycle, there is at least one $key \in r_i.out.Key$ such that $(key, RID(r_i), \perp) \in r_{i+1}.In$ forever.

First of all note that due to the assignment in Line 100, at some point in time $r_i.level \geq 1$ will hold. Second, assume there is an $i \in \{1, 2, \ldots, k\}$ such that $r_{(i-1 \bmod k)+1}.level \leq r_{(i \bmod k)+1}.level$. Since there is some triple $(key, RID(r_{(i-1 \bmod k)+1}), \perp)$ in $r_{(i \bmod k)+1}.In$, during TIMEOUT, $RL(r_{(i \bmod k)+1})$ will send a PING$(r_{(i \bmod k)+1}.ID, r_{(i \bmod k)+1}.level, r_{(i \bmod k)+1}.sinkRID, key)$ message to $RL(r_{(i-1 \bmod k)+1})$. Upon receipt, $RL(r_{(i-1 \bmod k)+1})$ will completely remove $r_{(i-1 \bmod k)+1}$ since $r_{(i-1 \bmod k)+1}.level < r_{(i \bmod k)+1}.level + 1$ (see Line 165), yielding a contradiction. However, $r_{(i-1 \bmod k)+1}.level > r_{(i \bmod k)+1}.level$ for all $i \in \{1, 2, \ldots, k\}$ is a contradiction as well. Thus, such a cycle cannot exist. All in all, we obtain a contradiction to our assumption that none of the relays in the cycle will be removed completely in finite time and the claim of the lemma is proven. □

**Lemma 6.32.** *For every relay $r$, $ls(r) = \times$ or in every state $S$, $cs_S(r)$ is probing.*

*Proof.* Consider an arbitrary relay $r$. Assume $ls(r) \neq \times$ (otherwise we are done). Now assume for contradiction that there is a state $S$ such that $cs_S(r)$ is not probing: i.e., the node owning $cs_S(r)$ is inactive, $cs_S(r).In = \emptyset$ and there is no relay $r'$ owned by $RL(r)$ such that $(key, \perp, cs_S(r)) \in r'.In$ for any key $key$. Note that for an inactive node, no element is ever added to $r''.In$ for any relay $r''$ owned by that node. Thus after $S$ there will never be a relay $r'$ owned by $RL(r)$ such that $(key, \perp, cs_S(r)) \in r'.In$ for any key $key$. Furthermore, $cs_S(r).Buf$ will eventually become empty and, as a result, $cs_S(r)$ will be completely removed in Line 137 upon TIMEOUT yielding a contradiction. Thus $cs_S(r)$ is probing in every state. □

**Lemma 6.33.** *For every relay $r$ such that $r.out.ID \neq \perp$ and $ls(r) \neq \times$, there is a relay $r'$ such that $r.out.ID = r'.ID$, $r'$ is alive forever and for some state $S$, $r'.In \neq \emptyset$ in every state $S' \geq S$.*

*Proof.* Consider an arbitrary relay $r$ such that $r.out.ID \neq \perp$. Furthermore assume that $ls(r) \neq \times$ (otherwise we are done). Note that by Lemma 6.32, in every state $S$, $cs_S(r)$ is probing. Furthermore, note that during TIMEOUT, $cs(r)$ will be deleted if $cs(r).out.ID$ does not contain the RID of some relay layer (see Line 103) or if $cs(r).out.Key = \emptyset$ (see Line 111). Furthermore note that during TIMEOUT, $RL(r)$ will send a PROBE() message via $cs(r)$ to $RL(r.out.ID)$. If at any point in time there is no relay $r'$ owned by $RL(r.out.ID)$ such that $r'.ID = r.out.ID$, $RL(r.out.ID)$ will respond with an OUT-RELAY-CLOSED($r.out.ID$) message to $r$ causing $cs(r)$ to be removed completely (and yielding a contradiction). Otherwise, there is a relay $r'$ owned by $RL(r.out.ID)$ such that $r'.ID = r.out.ID$ forever. Now we can apply Lemma 6.30, which implies that $r'$ is alive forever and that there is a state $S$ such that $r'.In \neq \emptyset$ in every state $S' \geq S$. This finishes the proof of the lemma. □

**Lemma 6.34.** *For every alive relay $r$ such that $r.out.ID \neq \perp$, $ls(r) = \times$ or there is a sequence of alive relays $(r_1 = r, r_2, r_3, \ldots, r_k)$ such that for all $i \in \{1, \ldots, k-1\}$, $r_{i+1}.ID = r_i.out.ID$ and $r_k.out.ID = \perp$ and for every $i \in \{2, \ldots, k\}$, $r_i$ is and remains alive forever.*

*Proof.* Consider an arbitrary relay $r$ such that $r.out.ID \neq \perp$. Furthermore, assume that $ls(r) \neq \times$ (otherwise we are done). Again, Lemma 6.32 implies that $cs(r)$ is probing in every state. Note that according to Lemma 6.33, $r.out.ID = r'.ID$ for some relay $r'$ that is and remains alive forever and for which there is a state $S$ such that $r'.In \neq \emptyset$ in every state $S' \geq S$. Applying Lemma 6.33 inductively, we obtain a sequence of relays $(r = r_1, r_2 = r', r_3, \ldots)$ such that $r_{i+1}.ID = r_i.out.ID$ for all $i \in \{1, 2, \ldots\}$ and these relays continue to be alive forever (*) and there is a state $S_x$ such that for every $i \in \{2, 3, \ldots\}$, $r_i.In \neq \emptyset$ in every state $S'_x \geq S_x$ (**). Now assume for contradiction that the above sequence of relays is infinite. Since there is only a finite number of relays, the sequence must contain a cycle in this case. According to (*), all relays in this cycle continue to be alive forever; according to (**), they cannot be merged. Since an alive relay is never removed completely before becoming dead first unless it is merged, all of these relays are not removed completely. However, this represents a contradiction to Lemma 6.31. Thus, the above sequence of relays must be finite: i.e., there is an index $k$ such that $r_k.out.ID = \perp$. All in all, the claim is proven. □

**Lemma 6.35.** *For every relay $r$ such that $ls(r) \neq \times$, for every triple $(key, \perp, r') \in r.In$ in some state $S$ for some key key and some relay $r'$, there is a state $S' > S$ such that there is no $(key, \perp, r'') \in cs_{S'}(r).In$ for any relay $r''$.*

*Proof.* Consider an arbitrary relay $r$ such that $ls(r) \neq \times$. Furthermore, consider an arbitrary triple $(key, \perp, r') \in cs_S(r).In$ in some state $S$ for some key $key$ and

some relay $r'$. Note that whenever a new triple of the form $(key, \perp, r''')$ is added to $cs(r).In$ then there must have existed a triple $(key, \perp, r'') \in cs(r).In$ before and $r'''$ must be the relay resulting from a merge of $r''$ and other relays. Thus, in every state $S'$, if $(key, \perp, r'') \in cs_{S'}(r).In$, then $r'' = cs_{S'}(r')$ must hold. Note that Property 8 will be satisfied for $r$ at some state $S_0 \geq S$ due to Line 120 and the fact that whenever a new triple is put into $r.In$ either $key$ is uniquely created (see Line 9) or the first parameter $key$ already existed as the first parameter of another triple that is removed before (see Lines 21–22 and the merge description). If $ls(r') = \times$, $(key, \perp, cs(r'))$ will eventually be removed and we are done (it is easy to check in the pseudocode that at any occasion at which a relay $r''$ is removed completely for reasons other than being merged, all such triples $(key, \perp, r'')$ are removed before if they exist). Otherwise, Lemma 6.34 implies that there is a sequence of alive relays $(r_1 = r', r_2, r_3, \ldots, r_k)$ such that for all $i \in \{1, \ldots, k-1\}$, $r_{i+1}.ID = r_i.out.ID$ and $r_k.out.ID = \perp$ and for all $i \in \{2, \ldots, k\}$, $r_i$ continues to be alive forever. Lemma 6.30 implies that there is a state $S_1 \geq S_0$ such that for every $i \in \{1, 2, \ldots, k-1\}$ there is a $key_i \in cs(r_i).out.Key$ such that $(key_i, RID(r_i), \perp) \in r_{i+1}.In$ in every state $S_1' \geq S_1$.

We now show that after some state $S_2 \geq S_1$, there will be no message $m$ that contains a relay parameter with key $key$ in any $r_i.Buf$ for any $i \in \{1, \ldots, k-1\}$. Note that after the state $S_1'$ such that all messages still in buffers in $S_1$ have been transmitted and processed, if for any $i \in \{1, \ldots, k-1\}$ $r_i.Buf$ contains a message $m$ that contains a relay parameter with key $key$, then this message will eventually be in $r_{i+1}.Buf$: Since the message was put into $r_i.Buf$ after $S_1$, it must have a valid header for $r_{i+1}$. Now since a message containing a relay parameter with a key $key$ is only created when the message is sent by the application and with a unique key $key$ and could only be created by $RL(r_1)$ because $key$ belongs to $RL(r_1)$, as soon as the message is received by $r_k$ (which will happen by induction), there will not be a message $m$ that contains a relay parameter with key $key$ in any $r_i.Buf$ for all $i \in \{1, \ldots, k-1\}$.

We now prove the following claim: there is a state $S_3 \geq S_2$ such that $(key, \perp, cs_{S_3}(r')) \notin cs_{S'}(r).In$ or in every state $S_3' \geq S_3$, there will be no relay $r''$ owned by $RL(r_k)$ such that $key \in r''.out.Key$. Due to the claim we have just proven and because any new message containing a relay parameter would acquire a new key for this relay parameter, $r_k$ will never receive a message containing a relay parameter with key $key$ after $S_2$. Thus any relay $r''$ ever owned by $RL(r_k)$ such that $key \in r''.out.Key$ must exist during $S_2$. If for such a relay $r''$, $ls(r'') = \times$, we are done. Otherwise, Lemma 6.32 implies that in every state $S'$ $cs_{S'}(r'')$ is probing. This implies that $RL(r_k)$ will put a TRANSMIT$((Keys, RID(r''.ID), r''.out.ID)$, PROBE$(ControlKeys, keySequence))$ message into $cs(r'').Buf$ such that $Keys = r''.out.Key$. There are two options: If this message $m$ does not have a valid header for any relay, $RL(r_k)$ will receive a NOT-AUTHORIZED$(Keys, r''.out.ID)$ or an OUT-RELAY-CLOSED$(r''.out.ID)$ message as a response and in any case there will be no relay $r''$ owned by $RL(r_k)$ such that $key \in r''.out.Key$ afterwards (in the former case, the key is removed from $cs(r'').out.Key$; in the latter case, $cs(r'')$

is removed completely). If $m$ does have a valid header for $r$, then because of Property 8 for $r$ (which holds as argued at the beginning of the proof), the relay that the message had a valid header for must be $cs(r)$. In that case, according to Line 21, $(key, \perp, cs(r'))$ will be removed from $cs(r).In$ and the claim is proven.

In the following, we assume that $(key, \perp, cs_{S'}(r')) \in cs_{S'}(r).In$ in every state $S'$ (otherwise the claim of the lemma follows). In that case, as we have just proven, there is a state $S_3 \geq S_2$ such that in every state $S_3' \geq S_3$, there will be no relay $r''$ owned by $RL(r_k)$ such that $key \in r''.out.Key$. Observe that as long as $(key, \perp, cs(r')) \in cs(r).In$, $cs(r')$ is probing by definition. Therefore, during some execution of TIMEOUT after $S_3$, $RL(r)$ will put a TRANSMIT$((Keys', RID(r'.ID), r'.out.ID)$, PROBE$(ControlKeys, (Keys')))$ message into $cs(r').Buf$ such that $key \in ControlKeys$ and $Keys' = cs(r').out.Key$. Recall that for every $i \in \{1, 2, \ldots, k-1\}$ there is a $key_i \in cs(r_i).out.Key$ such that there is a triple $(key', RID(r_i), \perp) \in r_{i+1}.In$ in every state after $S_1$. Thus, there is at least one $key' \in Keys'$ such that $(key', RID(r), \perp) \in r_2.In$.

After $S_3$, for every $i \in \{2, \ldots, k-1\}$, if $RL(r_i)$ receives a PROBE$(ControlKeys, (Keys_1', Keys_2', \ldots, Keys_i'))$ message with a valid header for $r_i$ such that $key \in ControlKeys$ and $key_i \in Keys_j'$ for every $j \in \{1, 2, \ldots, i-1\}$ then $RL(r_{i+1})$ will eventually receive a PROBE$(ControlKeys, (Keys_1', Keys_2', \ldots, Keys_i', Keys_{i+1}'))$ message with a valid header for $r_{i+1}$ such that $key \in ControlKeys$ and for every $j \in \{1, 2, \ldots, i\}$, $key_j' \in Keys_j'$. This is due to Lines 54–60 and the fact that there is no message that contains a relay parameter with key $key$ in any $cs(r_i).Buf$ for any $i \in \{1, \ldots, k-1\}$ and because $S_3 \geq S_1$. By induction we obtain that $RL(r_k)$ will eventually receive a PROBE$(ControlKeys, (Keys_1', Keys_2', \ldots, Keys_{k-1}'))$ message with a valid header for $r_k$ such that $key \in ControlKeys$ and $key_j' \in Keys_j'$ for every $j \in \{1, 2, \ldots, k-1\}$. Since $RL(r_k)$ does not own a relay $r''$ such that $key \in r''.out.Key$ after $S_3$, $RL(r_k)$ will send a PROBEFAIL$(key, (Keys_1', Keys_2', \ldots, Keys_{k-1}'))$ message to $RL(r_{k-1})$. According to Lines 69–74, upon receipt of this message, $RL(r_{k-1})$ will send a PROBEFAIL$(key, (Keys_1', Keys_2', \ldots, Keys_{k-2}'))$ message to $RL(r_{k-2})$ and so on, until finally, $RL(r)$ will receive a PROBEFAIL$(key, (Keys_1'))$ message and remove $(key, \perp, cs(r'))$ from $cs(r).In$ (see Line 77). This represents the desired contradiction. Thus, there is a state $S' \geq S$ such that $(key, \perp, cs_{S'}(r') \notin cs_{S'}(r).In)$ and as argued at the beginning of the proof, this finishes the proof of the lemma. □

Using Lemma 6.35, we can prove Theorem 6.19, which we now restate:

**Theorem 6.19.** *Every dead relay is eventually removed completely.*

*Proof.* Consider an arbitrary dead relay $r$. Note that since $r$ is dead, it will have $r.In = \emptyset$ in some state $S_1$ (due to Line 127 in the TIMEOUT action) and according to the pseudocode, no new element is added to $r.In$ after $S_1$. Since $r$ is dead, it does not accept any incoming messages (see Line 17) and thus does not forward any messages, nor is it possible to send a new message via $r$ (see Line 2). This implies two things: First, since all messages contained in $r.Buf$ during $S_1$ eventually have been transmitted and processed, there is a state $S_2 \geq S_1$ such that $r.Buf$ is empty.

Second, no new triple $(key, \perp, r) \in r'.In$ for any $key$ and any relay $r'$ can come into existence. Lemma 6.35 gives that all existing such ones will eventually be removed (either because $rl(r') = \times$ and the triple is removed upon the complete removal of $cs(r')$ or because of the claim of the lemma). Altogether, we obtain that $r$ will eventually be released. After this, $r$ will be removed completely in Line 134. □

Before we can also prove Theorem 6.20, we prove the following lemma, which states that relays of an inactive node do not exist forever as long as the application does not keep a relay or any of its successors alive forever:

**Lemma 6.36.** *If for every indirect relay $r$ there is a $r_s \in s(r)$ such that $r_s$ is deleted by the application, every relay $r$ owned by an inactive node will be removed completely in finite time.*

*Proof.* Assume that for every indirect relay $r'$ there is a $r'_s \in s(r')$ such that $r'_s$ is deleted by the application. Consider an arbitrary relay $r$ owned by an inactive node. If $r.out.ID = \perp$, $r$ will be removed completely in Line 137. Thus, in the following we assume $r.out.ID \neq \perp$ and assume that $r$ will not be removed completely in finite time (otherwise we are done). Our first goal is to show that $r.In$ will eventually be empty.

Note that for every relay $r'$ owned by an inactive node, each element is added to $r'.In$ only if it is of the form $(key, RID, \perp)$ and if it replaces a triple $(key, \perp, r'')$ in $r'.In$ for some relay $r''$ owned by $RL(r')$ (see Lines 21–22). Thus, Lemma 6.35 gives that eventually there will be no $(key, \perp, r'')$ in $r.In$ for any key $key$ and any relay $r''$ and no new elements will be added to $r.In$ (note that since the node owning $r$ is inactive, $cs_S(r) = r$ in every state $S$ after the node has become inactive). The TIMEOUT action ensures that from this point in time on, all elements in $r.In$ are of the form $(key, RID, \perp)$ for some $RID$. We now show that all of these triples will eventually be removed from $r.In$.

Consider an arbitrary triple $(key, RID, \perp) \in r.In$. Note that during TIMEOUT, $RL(r)$ regularly sends a PING$(r.ID, r.level, r.sinkRID, key)$ message to the relay layer $RL'$ with RID $RID$. According to the protocol, $RL'$ checks whether it has a relay $r'$ with $r'.out.ID = r.ID$. If not, according to the pseudocode (see Line 167), $RL'$ sends an IN-RELAY-CLOSED$(key, RID, r.ID)$ message to $RL(r)$, causing $r$ to remove $(key, RID, \perp)$ from $r.In$ and we are done. Otherwise, note that since $r.out.ID \neq \perp$, $r'$ must be indirect by definition. Thus, even if $r'$ is alive, there is a $r'_s \in s(r')$ such that $r'_s$ will be deleted at some point in time. Theorem 6.19 yields that $r'_s$ will also be removed completely at some point in time. After this has happened, $RL'$ will respond to the next PING$(r.ID, r.level, r.sinkRID, key)$ message after this with an IN-RELAY-CLOSED$(key, RID, r.ID)$ message to $RL(r)$, causing $r$ to remove $(key, RID, \perp)$ from $r.In$.

All in all, we have proven that $r.In$ will be empty at some state $S$ and in every subsequent state. Thus, as soon as all messages still in $r.Buf$ during $S$ have been

transmitted and processed, the check in Line 135 will eventually evaluate to true and $r$ will be removed completely in Line 137. □

We can now prove Theorem 6.20, which we recap as follows:

**Theorem 6.20.** *If for every indirect relay $r$ the application eventually deletes a successor of $r$, all relay layers of inactive nodes will eventually be shut down.*

*Proof.* Assume that for every indirect relay $r$ the application eventually deletes a successor of $r$. Consider an arbitrary inactive node $P$. Note that every relay owned by $P$ will be removed completely in finite time by Lemma 6.36. Thus, during some execution of TIMEOUT, the relay layer of $P$ will shut down completely in Line 152. Since $P$ was chosen arbitrarily, this finishes the proof of the theorem. □

### 6.3.3. Additional proofs for Theorem 6.21

Since we need multiple additional lemmas to prove Theorem 6.21, we further separate this subsection into three parts. Each part is finished when some significant result, called a *milestone*, has been proven. These milestones are as follows:

- **Milestone 1**: If the application is deliberate, every out-confirmed lingering relay $r$ remains out-confirmed as long as $r$ is lingering unless it is merged. If an out-confirmed relay is merged while it is out-confirmed, the resulting relay is out-confirmed as well. (Lemma 6.46)
- **Milestone 2**: If the application is deliberate, every valid relay $r$ remains valid as long as it is not deleted or merged with other relays. If it is merged, the resulting relay is valid. (Lemma 6.50)
- **Milestone 3**: If the application is deliberate, every message sent via a valid relay $r$ will be received by the sink node of $r$. (Theorem 6.21)

The motivation for these milestones is as follows: The final goal of this section is to prove that a message $m$ sent via a valid relay $r$ is correctly received by its sink node (Milestone 3). One major ingredient for this proof is that every relay $r'$ on the path from $r$ to its sink is valid as long as $m$ has not passed $r'$ yet. To prove this, we need a part of the claim of Milestone 2, namely that a valid relay remains valid unless it is deleted or merged (note that Milestone 2 is formulated in a more general way as we will rely on this lemma later on as well). The general idea to prove this is to show for every property of a valid relay $r$ that it remains satisfied as long as $r$ is not deleted or merged. Since Property 6 relies on another relay (the next relay on the path to the sink), we need to prove this via induction on the level of a relay. This is still not sufficient yet, though: Property 6c) (for the unconfirmed keys) has a possible dependency on relays owned by the same relay layer as $r$ that might even have a higher level than $r$. Fortunately, these relays do not need to be valid but only out-confirmed (simply put, it is irrelevant whether there is something wrong with their $.In$ sets). Thus, in Milestone 1 we show that these relays are out-confirmed as long as they are relevant for a key to be feasible.

### Milestone 1 Proofs

For Milestone 1, the first goal is to show that for many of the properties of a valid relay, once they hold, they hold forever, although for some of them we need to make some additional assumptions (this is because if for a pair of relays $r, r'$, $r'.ID = r.out.ID$ and $r'$ is not valid, it might, for example, cause $r.level$ to be changed which would then cause an existing PING() message sent by $r$ to have a *level* value different from $r.level$, thus invalidating Property 9). After that, we show that in a deliberate application, when a subset of the properties hold for a relay $r$, then no internal calls of **delete** $\hat{r}$ occur: i.e., $r$ can be deleted only when the application requests that. Subsequently, making use of these results we prove that every confirmed relay remains confirmed as long as its $.In$ set is not empty, which will be relevant to prove that an out-confirmed relay remains out-confirmed while it is lingering, which constitutes the first part of the claim of Milestone 1 (the second part of which can be proven directly).

We begin with a series of properties that remain valid once they are without any further conditions:

**Lemma 6.37.** *If any of the following properties holds for a relay $r$, this property continues to hold as long as $r$ exists (i.e., until $r$ is removed completely or forever if this does not happen): Property 2, Property 3, Property 4, Property 5, Property 7, Property 8 and Property 10.*

*Proof.* We prove the claim for each of the properties individually. Consider an arbitrary relay $r$ such that Property 2 holds for $r$. Note that $r.ID$ is never changed for any existing relay. Thus, Property 2 continues to hold.

Now consider an arbitrary relay $r$ such that Property 3 holds for $r$. Note that the form of $r.out$ is never changed for any existing relay. Thus Property 3 continues to hold.

Next consider an arbitrary relay $r$ such that Property 4 holds for $r$. Note that a new key is added to $r'.out.Key$ for any relay $r'$ only when the relay is created (see Line 45) and only if there is no other relay $r''$ with $key \in r''.out.Key$ (see Line 40). Thus, Property 4 continues to hold.

Next consider an arbitrary relay $r$ such that Property 5 holds for $r$. This implies that $r$ is a sink relay. Note that $r.out$ is never changed for a sink relay with $r.out.Key = \emptyset$. Additionally, check that $r.level$ is changed only if $r.out.ID = \perp$ and $r.level \neq 0$ or if $r.out.ID \neq \perp$. Furthermore, $r.sinkRID$ is changed only if $r.out.ID = \perp$ and $r.sinkRID \neq sinkRID(r)$ or $r.out.ID \neq \perp$. Should the node owning $r$ become inactive (i.e., it executes **stop**) then by definition, $r$ would be deleted immediately. Furthermore, note that should the node owning $r$ become inactive, $r$ would immediately be removed completely. Thus, as long as $r$ exists, Property 5 continues to hold.

Consider an arbitrary relay $r$ such that Property 7 holds for $r$. Note that if $(key, RID, \perp)$ is added to $r.In$ then $RID \neq \perp$. Furthermore, note that when a new $(key, \perp, r'')$ is added to $r.In$, then $r''$ is owned by $RL(r)$ and this only happens

in one of the following two cases: The first is that a relay reference is sent via $r''$ by the application and $r''$ is a non-sink relay. In this case, Property 7 continues to hold. The second is that an existing $(key, \perp, r''')$ is replaced by $(key, \perp, r'')$ because $r'''$ is merged into the new relay $r''$. If Property 7 held before, it will also hold after this. Note that according to the pseudocode if $r''$ is removed completely, every triple $(key, \perp, r'')$ is also removed from $r.In$. Since there are no other types of elements added to $r.In$ than the above mentioned, Property 7 continues to be fulfilled.

Now consider an arbitrary relay $r$ such that Property 8 holds for $r$. Note that the only occasion at which a key becomes the first parameter of an element in $r.In$ that has not been a first parameter of an element in $r.In$ before, is in Line 10 in which $key$ has just been generated as a locally unique $key$ belonging to $RID(r)$. Thus, Property 8 continues to be fulfilled.

Last consider an arbitrary relay $r$ such that Property 10 holds for $r$. Check in the pseudocode that the only occasion at which a NOT-AUTHORIZED$(Keys, r.ID)$ message is sent is in Line 64, in which case a message $m$ with header $(Keys, senderRID, r.ID)$ was received by $RL(r)$ and did not have a valid header for $r$, i.e., for no $key \in Keys$, $(key, senderRID, \perp \in r.In$ or $(key, \perp, r') \in r.In$ for some relay $r'$. Furthermore, note that whenever a new key $key$ is added as the first parameter of a triple to $r.In$, $key$ has just been created as a globally unique key. Thus, adding an element to $r.In$ also does not violate Property 10. All in all, Property 10 continues to hold for $r$. □

**Lemma 6.38.** *For every relay $r$ such that there is a relay $r'$ such that (i) $r'.ID = r.out.ID$, (ii) $r'$ is and remains valid, and (iii) Property 6b) holds for $r$, Property 6b) continues to hold for $r$ as long as $r$ exists.*

*Proof.* We prove the following claim by induction on the level of a relay: for every relay $r$, if (i) $r.out.ID = \perp$ and $r$ is and remains valid or (ii) $r.out.ID = r'$ for some relay $r'$ and $r'$ is and remains valid and Property 6b) holds for $r$, $r.level$ and $r.sinkRID$ do not change. We prove the claim by induction on the level of a relay. Note that since $r.level \in \mathbb{N}_0$ for every relay $r$, the induction base is given by $r.level = 0$.

Consider an arbitrary relay $r$ that fulfills the above conditions and such that $r.level = 0$. Note that if $r.out.ID = r'$ for some valid relay $r'$ and Property 6b) holds for $r$, $r.level > 0$ follows, which represents a contradiction. Thus, if $r.level = 0$, $r.out.ID = \perp$ must hold. In that case, according to the assumption, $r$ is and remains valid, so Property 5 will continue to hold and $r.level$ and $r.sinkRID$ will remain the same.

Now consider an arbitrary relay $r$ such that $r.level > 1$ and assume the claim holds for every relay $r'$ such that $r'.level < r.level$. By the above argument, we are then in case (ii): i.e., $r.out.ID = r'$ for some relay $r'$ and $r'$ is and remains valid and Property 6b) holds for $r$. By the induction hypothesis, $r'.level$ and $r'.sinkRID$ do not change. Assume for contradiction that $r.level$ or $r.sinkRID$ are changed.

This can only happen if $RL(r)$ receives a PING$(ID, level, sinkRID, key)$ message with $ID = r.out.ID$ and $r.level \neq level + 1$ or $r.sinkRID \neq sinkRID$, which would contradict either Property 9 for $r'$ or Property 6b) for $r$. Thus, $r.level$ and $r.sinkRID$ are never changed. This finishes the induction.

The claim of the lemma now follows immediately from the claim we just proved. □

**Lemma 6.39.** *For every relay $r$ such that Property 8 and Property 9 hold for $r$ and (i) Property 5 holds for $r$ or (ii) Property 6a) holds and continues to hold for $r$ and Property 6b) holds for $r$, Property 9 continues to hold for $r$ as long as $r$ exists.*

*Proof.* Consider an arbitrary relay $r$ such that Property 8 and Property 9 hold for $r$ and Property 5 holds for $r$ or Property 6a) holds and continues to hold for $r$ and Property 6b) holds for $r$. Note that Property 8 continues to hold according to Lemma 6.37.

We first consider the case $r.out.ID = \perp$. By assumption in this case, Property 5 holds for $r$. Lemma 6.37 implies that this property will continue to hold. Thus $r.level = 0$ and $r.sinkRID = RID(r)$ holds and will continue to hold. Now note that a PING$(r.ID, level, sinkRID, key)$ message is sent only during TIMEOUT and only with $level = r.level$ and $sinkRID = r.sinkRID$ and $key$ such that $(key, RID, \perp) \in r.In$. By Property 8 the latter prevents $(key, \perp, r'') \in r.In$ to hold. Thus, no PING() message contradicting Property 9 is sent.

Now consider the case $r.out.ID \neq \perp$. By assumption, there is a relay $r'$ such that $r'.ID = r.out.ID$ and $r'$ is and remains valid and Property 6b) holds for $r$. Note that according to Lemma 6.38, Property 6b) continues to hold for $r$. From the proof of that lemma, we know that $r'.level$ and $r'.sinkRID$ are not changed then. Again, observe that a PING$(r.ID, level, sinkRID, key)$ message is sent only during TIMEOUT and only with $level = r.level$ and $sinkRID = r.sinkRID$ and $key$ such that $(key, RID, \perp) \in r.In$. Again, by Property 8 the latter prevents $(key, \perp, r'') \in r.In$ to hold. Thus, no PING() message contradicting Property 9 is sent in this case as well. □

We now prove a lemma that implies that the internal calls of **delete** $\hat{r}$ according to the pseudocode do not occur on relays satisfying certain conditions when the application is deliberate:

**Lemma 6.40.** *If the application is deliberate then for every alive relay $r$ such that (i) Properties 2–4 hold for $r$ and (ii) $r.out.ID = \perp$ or Property 6 holds for $r$, then **delete** $r$ is only called when $r$ is deleted by the application.*

*Proof.* Assume that the application is deliberate. Let $r$ be an alive relay such that Properties 2–4 hold for $r$ and $r.out.ID = \perp$ or Property 6 holds for $r$. We check all lines that contain a call of **delete**:

1. Line 87: If this line is executed, this means that the relay layer owning $r$ has received a NOT-AUTHORIZED$(Keys, outID)$ message with $r.out.ID =$

$outID \neq \perp$ and $r.out.Key \subseteq Keys$. Note that since $r.out.ID \neq \perp$, Property 6 holds for $r$ in this case: i.e., there is a valid relay $r'$ such that $r'.ID = r.out.ID$. Due to Property 6c) and the fact that Property 10 holds for $r'$ (since $r'$ is valid), we have a contradiction, so this call of **delete** cannot occur.

2. Line 92: If this line is executed, Property 2 for $r$ must have been violated, yielding a contradiction.

3. Line 103: If this line is executed, Property 3 must have been violated or $r.out.ID \neq \perp$ and there is no relay $r'$ such that $r'.ID = r.out.ID$, yielding a contradiction.

4. Line 111: If this line is executed, Property 6c) must have been violated, yielding a contradiction (note again that out-confirmed relays also satisfy this property).

5. Line 140: If this line is executed, Property 4 must have been violated before, yielding a contradiction.

6. Line 160: If this line is executed, $r.out.ID \neq \perp$ and Property 6b) for $r$ or Property 9 for the relay $r'$ with $r'.ID = r.out.ID$ must have been violated, yielding a contradiction to the fact that $r'$ is valid (implied by Property 6a).

7. Line 185: If this line is executed, $r.out.ID \neq \perp$ and Property 1 for the relay $r'$ with $r'.ID = r.out.ID$ must have been violated, yielding a contradiction to the fact that $r'$ is valid (which is implied by Property 6a).

All in all, $r$ is never deleted unless it is deleted by the application. This finishes the proof. □

Observe that Lemma 6.37, Lemma 6.39 and Lemma 6.40 directly imply the following corollary:

**Corollary 6.41.** *If the application is deliberate, every confirmed sink relay $r$ remains confirmed unless it is deleted by the application.*

Before we can prove a similar result for general confirmed relays, we need to prove the following claim:

**Lemma 6.42.** *For every out-confirmed lingering relay $r$ the following holds: as long as $r$ exists, the relay $r'$ such that $r'.ID = r.out.ID$ remains confirmed, $r$ remains lingering, and $r$ remains out-confirmed.*

*Proof.* Consider an arbitrary out-confirmed relay $r$ and assume that relay $r'$ such that $r'.ID = r.out.ID$ remains confirmed and that $r$ is and remains lingering. Note that Property 2, Property 3 and Property 4 continue to hold for $r$ according to Lemma 6.37. By assumption, Property 6a) continues to hold. Furthermore, note

that Property 6b) continues to hold as long as $r$ is alive according to Lemma 6.38. Thus, all that needs to be shown is that there continues to be a $key \in r.out.Key$ such that there is no IN-RELAY-CLOSED$(Keys, RID(r), r.out.ID)$ message in transit to $RL(r')$ such that $key \in Keys$ and $(key, RID(r), \bot) \in r'.In$ (note that this also implies Property 6c).

First of all, check that a $key$ is removed from $r.out.Key$ only if $RL(r)$ receives a NOT-AUTHORIZED$(Keys, r.out.ID)$ message with $key \in Keys$. However, due to Property 10 for $r'$, such a message cannot be sent for any $(key, RID(r), \bot) \in r'.In$.

Second, note that an IN-RELAY-CLOSED$(key, RID(r), r.out.ID)$ message is sent by $RL(r)$ only and only at two occasions: in Line 132, which is executed only if $r$ is dead, there is no relay $r''$ such that $(key, \bot, r) \in r''.In$, and $r.Buf = \emptyset$, which cannot be the case according to the assumption that $r$ is lingering; or in Line 167 if $RL(r)$ received a PING$(r.out.ID, level, sinkRID, key)$ message such that $key \notin r.out.Key$. Thus, for every existing $key \in r.out.Key$, no IN-RELAY-CLOSED$(key, RID(r), r.out.ID)$ message will be created.

Now consider the case that the last triple $(key, RID(r), \bot) \in r'.In$ such that $key \in r.out.Key$ and there is no IN-RELAY-CLOSED$(Keys, RID(r), r.out.ID)$ message in transit to $RL(r')$ is removed from $r'.In$. According to the pseudocode and the fact that $r'$ is confirmed, such a triple is only removed when $RL(r')$ receives an IN-RELAY-CLOSED$(Keys, RID(r))$ message with $key \in Keys$, which would represent a contradiction. Thus, $r$ remains out-confirmed. □

Now we can prove that every confirmed relay (and not only the confirmed sink relays) remains confirmed as long as certain conditions are fulfilled:

**Lemma 6.43.** *If the application is deliberate, every confirmed relay $r$ remains confirmed as long as $r.In \neq \emptyset$ and $r$ is not deleted by the application.*

*Proof.* For this proof, assume that the application is deliberate. We prove the claim by induction on the level of a relay (i.e., for a relay $r$ its value of $r.level$). Note that due to Property 5, Property 6b) and the fact that for every relay $r'$, $r'.level \in \mathbb{N}_0$, for every confirmed relay $r$, $r.level = 0$ if and only if $r.out.ID = \bot$. Furthermore, for every confirmed relay $r$, if $r.level \neq 0$ then $r.level > 0$ and $r.out.ID = r'.ID$ for some confirmed relay $r'$ such that $r'.level = r.level - 1$.

The induction base is given by Corollary 6.41. Thus, consider an arbitrary confirmed relay $r$ such that $r.level > 0$ and assume that the claim holds for all relays of level $r.level - 1$. Furthermore, assume that $r$ is not deleted by the application and that $r.In \neq \emptyset$ (otherwise, we are done). This implies that $r$ cannot be merged (since $r.In = \emptyset$ is a requirement for a merge). Note that Lemma 6.40 implies that $r$ is not deleted at all, which means that $r$ cannot be removed completely as long as $r.In \neq \emptyset$ (observe in the pseudocode that a non-sink relay $r'$ is never removed completely as long as $r'.In \neq \emptyset$ and $r'$ is alive). Let $r'$ be the relay such that $r'.ID = r.out.ID$. First of all, assume for contradiction that $r'$ is deleted when $r$ is still confirmed. By definition of a confirmed relay $r'.In \neq \emptyset$ at that point in time. Since the application is deliberate, it did not delete $r'$ then.

According to Lemma 6.40 (which can be applied because if $r'.out.ID \neq \perp$, $r'$ is out-confirmed by definition), $r'$ cannot have been deleted at all then, yielding the desired contradiction. Second, assume for contradiction that $r'$ is merged while $r$ is still confirmed. Again, by definition of a confirmed relay $r'.In \neq \emptyset$ at that point in time. Thus $r'$ cannot have been merged. Thus, in the following by the induction hypothesis and the fact that $r'.level = r.level - 1$, we obtain that $r'$ remains confirmed forever. Thus we can apply Lemma 6.42 (recall that every alive relay is lingering by definition and that every confirmed relay is out-confirmed as well), implying that as long as $r$ fulfills the above prerequisites, $r$ fulfills the definition of an out-confirmed relay. Together with Lemma 6.37, Lemma 6.38 and Lemma 6.39, we obtain that $r$ remains confirmed as long as $r$ fulfills the above prerequisites. □

**Lemma 6.44.** *For every relay $r$ the following holds: as long as $r$ is (i) lingering-valid or (ii) lingering and out-confirmed, $r$ is not removed completely unless it is merged.*

*Proof.* Consider an arbitrary relay $r$ that is lingering-valid or lingering and out-confirmed and assume that $r$ will not be merged. We consider all occasions at which a relay is removed completely:

1. When a relay is merged, but this does not happen to $r$ by assumption.
2. In Line 88, in which case $r.out.Key = \emptyset$, contradicting the assumption that $r$ is lingering-valid or out-confirmed (c.f. Property 6c).
3. In Line 96 in which case Property 2 would have been violated.
4. In Line 107, in which case Property 3 would have been violated or there would be no relay $r'$ such that $r'.ID = r.out.ID$.
5. In Line 115, in which case $r.out.Key = \emptyset$, contradicting the assumption that $r$ is lingering-valid or out-confirmed (c.f. Property 6c).
6. In Line 129, but this would require $r.out.ID = \perp$ (due to the check in Line 128).
7. In Line 134, but for this line to be executed $r.state \neq alive$ must hold (due to the check in Line 126), $r.Buf = \emptyset$ must hold and there must not be a relay $r'$ owned by $RL(r)$ such that $(key, \perp, r) \in r'.In$ (the last two facts are due to the check in Lines 130–131). This contradicts the assumption that $r$ is lingering.
8. In Line 137, but the preceding check makes sure that this line is not executed on a lingering relay.
9. In Line 144, but in this case Property 4 would have been violated.

10. In Line 165, but this cannot be the case, due to Property 6b) and the fact that Property 9 must hold for the relay $r'$ such that $r'.ID = r.out.ID$ according to the fact that $r$ is lingering-valid or out-confirmed.

11. In Line 178, but this line is only executed for sink relays, so it cannot affect $r$ as $r$ is lingering.

12. In Line 186, but in this case Property 1 would have been violated for the relay $r'$ such that $r'.ID = r.out.ID$, which would contradict the fact that $r$ is lingering-valid or out-confirmed.

All in all, $r$ is not removed completely. □

**Lemma 6.45.** *Whenever a relay $r$ is merged, then the resulting relay $r'$ satisfies Properties 1–4 and Properties 7–10. If any of Property 6a), Property 6b) or Property 6c) held for $r$ before the merge, the corresponding properties also hold for $r'$ after the merge.*

*Proof.* Consider an arbitrary relay $r$ and assume $r$ is merged into some new relay $r'$. Note that $r'.ID$ is a new and globally unique ID and that $r'.state = alive$. Thus Property 1 holds for the resulting relay $r'$. Furthermore, $r'.out = (Key, ID)$ and for every $key \in r'.out.Key$ there is no relay $r'' \neq r'$ owned by the same node such that $key \in r''.out.Key$, because otherwise the merge would not have taken place and because all relays that are merged together are completely removed immediately after that. Therefore, Properties2–4 hold for $r'$. Property 7, Property 8 and Property 10 of a valid relay follow from the fact that $r'.In = \emptyset$. Property 9 of a valid relay holds for $r$ also because $r'.ID$ was chosen globally uniquely. Since $r'.out.ID = r.out.ID$, Property 6a) holds for $r'$ if it held for $r$. Since $r'.level = r.level$ and $r'.sinkRID = r.sinkdRID$, the same applies to Property 6b). Since $r.out.Key \subseteq r'.out.Key$, Property 6c) holds for $r'$ if it held for $r$. This finishes the proof of the lemma. □

**Lemma 6.46.** *If the application is deliberate, every out-confirmed lingering relay $r$ remains out-confirmed as long as $r$ is lingering unless it is merged. If an out-confirmed relay is merged while it is out-confirmed, the resulting relay is out-confirmed as well.*

*Proof.* Assume that the application is deliberate. Consider an arbitrary out-confirmed lingering relay $r$. We begin with the first part of the claim. Note that as long as $r$ is lingering and not merged, it is not removed completely according to Lemma 6.44. Note that the relay $r'$ such that $r'.ID = r.out.ID$ must be confirmed initially by definition.

Assume for contradiction that $r'$ ceases to be confirmed at a point in time right before which $r$ was still out-confirmed. Note that as long as $r$ is out-confirmed, $r'.In \neq \emptyset$ by definition. Thus the fact that the application is deliberate and Lemma 6.40 imply that $r'$ cannot have ceased to be confirmed by being deleted.

Then, however, Lemma 6.43 implies that $r'$ cannot have ceased to be confirmed at all, yielding a contradiction. So when $r$ ceases to be out-confirmed, $r'$ was still confirmed right before.

Now assume that $r$ ceases to be out-confirmed at a point in time at which $r'$ was still confirmed right before. Furthermore, assume that $r$ is still lingering at that time. Then, Lemma 6.42 implies that $r$ cannot have ceased to be out-confirmed. Thus, all in all, $r$ remains out-confirmed as long as it is lingering.

We now prove the second part of the claim: Consider an arbitrary out-confirmed relay $r$ that is merged into a new relay $r'$. Lemma 6.45 implies that Properties 2–4 and Property 6 hold for $r$ as well. Even more, since $r'.out.ID$ is equal to $r.out.ID$ before the merge, the relay $r''$ such that $r'.out.ID = r''.ID$ is confirmed. Since $r''.In$ is unaffected by the merge and since $r.out.Key \subseteq r'.out.Key$, there is also a key feasible for $r'$ such that $(key, RID(r'), \bot) \in r''.In$ (since $r$ was out-confirmed). Altogether $r'$ is out-confirmed as well. □

### Milestone 2 Proofs

To prove that valid relays and their successors remain valid unless they are deleted, we first prove that feasible keys remain feasible as long as certain conditions are fulfilled, which is required for Property 6c). We then briefly prove that Property 1 continues to hold for an alive relay, which completes the set of "if Property X is satisfied, it will continue to be satisfied" proofs required to carry out the induction that proves the main claim of Milestone 2: that every valid relay remains valid as long as it is not deleted and that any merge of a valid relay results in a valid relay.

As mentioned, we begin with a result about feasible keys:

**Lemma 6.47.** *If the application is deliberate then every key key that is feasible for a relay $r$ such that Properties 2–4, Property 6a) and Property 6b) hold for $r$ remains feasible for $r$ as long as: (i) $r$ exists, (ii) the relay $r'$ such that $r'.ID = r.out.ID$ is valid, and (iii) $r$ is lingering.*

*Proof.* Assume that the application is deliberate. Consider an arbitrary relay $r$ such that Properties 2–4, Property 6a) and Property 6b) hold for $r$. Furthermore, assume in the following that $r$ is not removed completely, that the relay $r'$ such that $r'.ID = r.out.ID$ remains valid and that $r$ is lingering (as soon as this is not the case, there is nothing to be proven any more). Note that Properties 2–4 continue to be satisfied then according to Lemma 6.37. Furthermore, consider an arbitrary key $key$ that is feasible for $r$.

First of all check that $key$ could be removed from $r.out.Key$ only if $RL(r)$ receives a NOT-AUTHORIZED$(Keys, r.out.ID)$ message with $key \in Keys$. Since $key$ is feasible for $r$, such a message cannot have existed, due to Property 10 for $r'$ and the fact that $r'$ is valid.

Next, note that an IN-RELAY-CLOSED$(key, RID(r), r.out.ID)$ message could only be sent by $RL(r)$ and only if $r$ is dead, there is no relay $r''$ such that

$(key, \perp, r) \in r''.In$, and $r.Buf = \emptyset$ (see Line 132), which cannot be the case according to the assumption that $r$ is lingering, or if $RL(r)$ received a PING$(ID, level, sinkRID, key)$ message and $key \notin r.out.Key$ (see Line 167), which would represent a contradiction. Thus, no IN-RELAY-CLOSED$(key, RID(r), r.out.ID)$ message will be created.

Now assume that $(key, RID(r), \perp) \in r'.In$ (i.e., $key$ is a confirmed key). Note that this triple is removed from $r'.In$ only when $RL(r')$ receives a message IN-RELAY-CLOSED$(Keys, RID(r), r'.ID)$ with $key \in Keys$. Thus, in this case $key$ remains feasible.

Last, assume that $key$ is feasible according to the second case of the definition of a feasible key. In this case, $(key, \perp, r'') \in r'.In$ for some out-confirmed $r''$ such that $r''.sinkRID = RID(r)$. Note that since $r''$ is lingering by definition, it remains out-confirmed according to Lemma 6.46 as long as it is not merged (in which case the triple would be removed, which is considered below). We first show that no PROBEFAIL() or PROBE() messages contradicting the definition are created. Note that any PROBEFAIL$(key, keySequence)$ message is sent only upon receipt of a PROBE$(ControlKeys, keqSequence)$ message such that $key \in ControlKeys$ received by a sink relay $r_s$ and only if $RL(r_s)$ does not own a relay $r'_s$ such that $key \in r'_s.out.Key$ (see Line 35). Due to the fact that for the sequence of relays $(r_1 = r'', r_2, \ldots, r_k)$ such that $r_{i+1}.ID = r_i.out.ID$ for all $1 \leq i < k$ and $r_k.out.ID = \perp$ there is no PROBE$(ControlKeys, keySequence)$ message such that $key \in ControlKeys$ in $r'''.Buf$ for any relay $r''' \notin \{r_1, \ldots, r_{k-1}\}$ by assumption, the only sink relay that could receive such a message is $r_k$. Due to the fact that $r''$ is out-confirmed, Property 6b) and the fact that $r''.sinkRID = RID(r)$, $RL(r_k) = RL(r)$, a PROBEFAIL$(key, keySequence)$ message thus cannot be created. Next, note that a PROBE$(ControlKeys, keySequence)$ message is created only either by $RL(r'')$ and put into $r''.Buf$ (see Line 149) or when a non-sink relay receives a PROBE$(ControlKeys', keySequence)$ message such that $ControlKeys \subseteq ControlKeys'$ (see Line 60). By the above assumption, again, this can only be a relay $r_i$ for $2 \leq k-1$, which is confirmed by the assumption that $r''$ is out-confirmed. Hence the new PROBE$(ControlKeys, keySequence)$ message is put into $r_i.Buf$, thus not contradicting the definition of a feasible key. Last assume that $(key, \perp, r'')$ is removed from $r'.In$. According to the pseudocode there are four possibilities for this: in the first case, the triple is replaced by a triple $(key, RID, \perp)$ such that $RID = r''.sinkRID = RID(r)$ (see Lines 19–22), in which case $key$ remains to be feasible. The second case is that $RL(r')$ receives a PROBEFAIL$(key, keySequence)$ message, which cannot have existed before if $key$ was feasible before. The third case is that $r''$ is merged into some relay $r'''$, in which case $(key, \perp, r'')$ is replaced by $(key, \perp, r''')$ in $r'.In$. According to Lemma 6.46, since $r''$ was out-confirmed by the definition of $key$, $r'''$ is out-confirmed as well and the property continues to hold. The fourth case is that $r''$ is removed completely for reasons other than being merged, but this cannot be the case according to Lemma 6.44 and the fact that $r''$ is lingering and out-confirmed.

All in all, in any possible case, $key$ remains feasible as long as (i) $r$ exists, (ii)

the relay $r'$ such that $r'.ID = r.out.ID$ is valid, and (iii) $r$ is lingering. □

**Lemma 6.48.** *For every relay $r$ such that Property 1 holds for $r$, Property 1 continues to hold for $r$ as long as $r$ is alive.*

*Proof.* Consider an arbitrary relay $r$ such that Property 1 holds for $r$. Note that there are two occasions at which an OUT-RELAY-CLOSED($r.ID$) message is sent: in Line 67 and in Line 175. In both cases, the relay with ID $r.ID$ either does not exist or is dead. Thus, as long as $r$ is not deleted, Property 1 continues to hold. □

Lemma 6.37, Lemma 6.39, Lemma 6.40, Lemma 6.45 and Lemma 6.48 together directly imply the following corollary:

**Corollary 6.49.** *Every valid sink relay $r$ remains valid unless it is deleted by the application.*

**Lemma 6.50.** *If the application is deliberate, every valid relay $r$ remains valid as long as it is not deleted by the application or merged with other relays. If it is merged, the resulting relay is valid.*

*Proof.* For this proof, assume that the application is deliberate. Note that the fact that when a valid relay is merged the resulting relay is valid follows directly from Lemma 6.45. Thus, all that needs to be proven is that every valid relay $r$ remains valid as long as it is not deleted or merged with other relays.

We prove this claim by induction on the level of a relay (i.e., for a relay $r$ its value of $r.level$). Note that due to Property 5, Property 6a), Property 6b) and the fact that $r'.level \in \mathbb{N}_0$ for every relay $r'$, for every valid relay $r$, $r.level = 0$ if and only if $r.out.ID = \perp$. Furthermore, for every valid relay $r$, if $r.level \neq 0$ then $r.level > 0$ and $r.out.ID = r'.ID$ for some valid relay $r'$ such that $r'.level = r.level - 1$.

The induction base is given by Corollary 6.49. Thus, consider an arbitrary valid relay $r$ such that $r.level > 0$ and assume that the claim holds for all relays of level $r.level - 1$. Note that if $r$ is deleted by the application while it is still valid, we are done. Thus, assume in the following that $r$ is not deleted by the application, which by Lemma 6.40 implies that $r$ is not deleted at all. Furthermore, note that if $r$ is merged while it is still valid, the resulting relay is valid according to Lemma 6.45. Thus, we consider all possible causes for $r$ to cease being valid without being deleted or merged. Property 1 cannot become unfulfilled as long as $r$ is alive according to Lemma 6.48. Furthermore, Properties 2–4, Property 7, Property 8 and Property 10 cannot become unfulfilled at all as long as $r$ exists according to Lemma 6.37. Note that as long as $r'$ is not deleted or merged, $r'$ remains valid by the induction hypothesis and the fact that $r'.level = r.level - 1$. As long as this is the case, Property 6a) remains fulfilled, Property 6b) continues to hold according to Lemma 6.38, and Property 9 continues to hold true according to Lemma 6.39 (note that $r$ is not removed completely according to the pseudocode if it is neither deleted nor merged). Property 6c) is then satisfied due to Lemma 6.47 (note that $r$ is lingering as long as it is alive). All in all, if $r$ stops being valid, $r'$ must have

been deleted or merged first. Thus, assume that $r'$ is deleted or merged when $r$ is still valid. According to Property 6c), $r'.In \neq \emptyset$ while $r$ is valid. This implies that $r'$ cannot be merged as long as $r$ is valid. Since the application is deliberate, $r'$ cannot be deleted by the application. Recall that $r'$ remains valid as long as it is not deleted. Hence, we can apply Lemma 6.40, yielding that $r'$ is not deleted at all. Thus, $r'$ remains valid and (as argued before) $r$ also remains valid. This finishes the induction step.

All in all, we obtain the claim of the lemma. □

### Milestone 3 Proofs

For Milestone 3 we want to show that messages sent via valid relays are correctly delivered to their sink nodes. Note that the fact that valid relays remain valid, as we proved in Milestone 2, is not sufficient yet for the following reason: So far, we considered that the application is deliberate, i.e., it does not delete a relay with incoming connections. As soon as a relay no longer has any incoming connections, however, it may freely be deleted. This could have the effect that even though a message $m$ is put into the buffer of a valid relay $r$, $r$ is deleted before $m$ has been transmitted and processed. To formally verify that the message is delivered correctly to the sink anyway, we prove that $r$ will remain to be lingering-valid as long as $m$ is still in its buffer. After that, we conclude with the main result of Milestone 3 and this section.

As a first step to prove that lingering-valid relays remain lingering-valid as long as they exist, we prove that under certain conditions, some of the properties of a valid relay required for a lingering-valid relay continue to be satisfied for a relay $r$ even if $r$ does not necessarily stay alive.

**Lemma 6.51.** *If the application is deliberate, for every relay $r$ such that Properties 2–4, Property 6a) and Property 6b) hold for $r$, the following holds: as long as $r$ exists, the relay $r'$ such that $r'.ID = r.out.ID$ is valid and $r$ is lingering, the above properties continue to hold for $r$.*

*Proof.* Assume that the application is deliberate. Consider an arbitrary relay $r$ such that Properties 2–4, Property 6a), and Property 6b) hold for $r$. This implies that $r$ is not a sink: i.e., there is a relay $r'$ such that $r.out.ID = r'.ID \neq \bot$. Furthermore, assume in the following that $r$ will not be removed completely, the relay $r'$ such that $r'.ID = r.out.ID$ remains valid, and that $r$ is lingering (as soon as this is not the case, there is nothing to be proven any more). Note that Properties 2–4 then continue to hold according to Lemma 6.37. Since $r.out.ID$ is never changed, $r.out.ID \neq \bot$ will hold forever. This implies that as long as $r'$ is valid, Property 6a) continues to hold. Lemma 6.38 implies that also Property 6b) continues to hold. □

Note that Lemma 6.47 and Lemma 6.51 imply the following corollary:

**Corollary 6.52.** *If the application is deliberate, for every relay $r$ such that Properties 2–4 and Property 6 hold for $r$, the following holds: as long as $r$ exists, the relay $r'$ such that $r'.ID = r.out.ID$ remains valid, $r$ is lingering, and Property 6 continues to hold for $r$.*

**Lemma 6.53.** *If the application is deliberate, every lingering-valid relay remains lingering-valid unless it is released or merged with other relays. If it is merged before it is released, the resulting relay is valid.*

*Proof.* Assume that the application is deliberate. Consider an arbitrary lingering-valid relay $r$. We first show that as long as $r$ is not released or merged, $r$ remains lingering-valid. Thus assume for now that $r$ is not released or merged. Note that if $r$ is removed completely, it is released right before and we are done in this case. Thus assume that $r$ is not removed completely. Lemma 6.37 implies that Properties 2, 3, 4, 7, 8 and 10 remain valid for $r$. Since Property 6 holds for $r$, there is a valid relay $r'$ such that $r'.ID = r.out.ID$. Property 6c) particularly implies that $r'.In \neq \emptyset$. Thus, $r'$ cannot be merged as long as $r$ is lingering-valid. Together with Lemma 6.50, we obtain that $r'$ remains valid as long as $r$ is lingering-valid. Next, Corollary 6.52 implies that Property 6 remains valid for $r$. Therefore, Lemma 6.39 can be applied, yielding that Property 9 remains valid for $r$. All in all, $r$ remains lingering-valid.

Now assume that $r$ is merged before it is released. This implies that $r.alive = true$ and $r.In = \emptyset$ (otherwise $r$ could not be merged). Thus Property 1 holds for the resulting relay $r'$ (note that its ID is newly created). Note that Properties 2–4 follow from the way the **merge** command works. Property 6 must hold for $r'$ since it held for $r$. Property 7 and Property 8 of a valid relay follow from the fact that $r'.In = \emptyset$. Property 9 of a valid relay holds for $r$ because $r'.ID$ was chosen globally uniquely. Last, Property 10 holds because $r'.In = \emptyset$. Thus, $r'$ is a valid relay.

This finishes the proof of the lemma. □

Equipped with all these results, we are ready to prove Theorem 6.21, which we recall as follows:

**Theorem 6.21.** *If the application is deliberate, every message sent via a valid relay $r$ will be received by the sink node of $r$.*

*Proof.* Assume the application is deliberate and consider an arbitrary valid relay $r$ and an arbitrary message $m$ such that **send($\hat{r}$, m)** is called on $RL(r)$. Observe that the fact that $r$ is valid implies that there is a sequence of valid relays $(r = r_1, r_2, \ldots, r_k)$ such that $r_{i+1}.ID = r_i.out.ID$ for all $i \in \{1, \ldots, k-1\}$, $r_k.out.ID = \perp$, and $RID(r_k) = r.sinkRID$. The assumption that the application is deliberate and Lemma 6.53 imply for every $i \in \{1, 2, \ldots, k-1\}$ that $r_{i+1}$ continues to exist and to be valid as long as $r_i$ is lingering because $cs(r_i)$ then also remains valid or lingering-valid. Thus $r_{i+1}.In \neq \emptyset$, implying that $r_{i+1}$ cannot be merged (*). We now show that if $RL(r_i)$ puts $m$ into $r_i.Buf$ for any $i \in \{1, \ldots, k-1\}$, it will be received with a valid header by $RL(r_{i+1})$. Since $m$ is initially put into

$r_1.Buf$ and because $RL(r_k)$ will forward $m$ to the node owning $r_k$ if it receives the message with a valid header for $r_k$, this is sufficient to prove the theorem.

Let $i \in \{1, \ldots, k-1\}$ be arbitrary but fixed. Assume $RL(r_i)$ puts $m$ into $r_i.Buf$. According to the pseudocode (c.f. Line 14 and Line 60), the header used for this message is $(Keys, RID(r_i.ID), r_i.out.ID)$, in which $Keys = r_i.out.Key$. Due to Property 6 for $r_i$, $(Keys, RID(r_i.ID), r_i.out.ID)$ is initially a valid message header. Thus, all we still need to prove is that it remains valid until it is received. Note that $r_i$ must be alive when $m' := \textsc{Transmit}((Keys, RID(r_i.ID), r_i.out.ID), m)$ is put into $r_i.Buf$ and as long as $m' \in r_i.Buf$, $r_i$ is lingering. According to (*), $r_{i+1}$ continues to exist and to be valid as long as the message has not been received by $RL(r_{i+1})$. This allows us to apply Lemma 6.47, yielding that every feasible key in $Keys$ remains feasible until $m'$ is received. Since there was at least one feasible key in $Keys$ when the message was sent due to Property 6c) for $r_i$, $m$ will be received with a valid header by $RL(r_{i+1})$ and the proof is complete. □

### 6.3.4. Additional Proofs for Theorem 6.22

In this subsection, we provide some additional lemmas and their proofs that are required to prove the closure property of self-stabilization. First of all, we prove that dead-valid relays remain dead-valid until they are removed completely. After that we obtain that every valid deleted relay immediately becomes a dead-valid relay (unless it is removed immediately). We then show that relays created from valid relay parameters are valid relays and that relay parameters created from a valid relay and sent via a valid relay are valid and remain as such until they reach the sink. Putting all these pieces together, we obtain that computations starting in legal states consist of legal states only.

We begin with proving the following lemma, which is proven very similarly to Lemma 6.53:

**Lemma 6.54.** *If the application is deliberate, every dead-valid relay remains dead-valid as long as it exists.*

*Proof.* Assume that the application is deliberate. Consider an arbitrary dead-valid relay $r$. Lemma 6.37 implies that Properties 2, 3, 4, 7, 8 and 10 remain valid for $r$. Since Property 6 holds for $r$, there is a valid relay $r'$ such that $r'.ID = r.out.ID$. Property 6c) particularly implies that $r'.In \neq \emptyset$. Thus, $r'$ cannot be merged and Lemma 6.50 together with the fact that the application is deliberate yields that $r'$ remains being valid. Next, Corollary 6.52 implies that Property 6 remains valid for $r$. Therefore, Lemma 6.39 can be applied, yielding that Property 9 remains valid for $r$. All in all, $r$ remains dead-valid. □

From the pseudocode and Lemma 6.54, we immediately obtain the following corollary:

**Corollary 6.55.** *If a valid relay $r$ is deleted by the application, then it is removed completely immediately or immediately becomes dead-valid and remains dead-valid until it is removed completely.*

**Lemma 6.56.** *If the application is deliberate, each relay created from a valid relay parameter is a valid relay.*

*Proof.* Assume a relay $r$ is created from a valid relay parameter $(key, ID, level, sinkRID)$. This happens when the message $m = ((Keys, senderRID, outID), action(parameters))$ containing the valid relay parameter is received by a relay layer $RL(outID)$ such that the relay with ID $outID$ is a sink. Note that $r$ is created with $r.state = alive$, a globally unique $r.ID$, a pair $r.out = (Key, ID)$ such that $Key$ is a set and empty $r.In$. Thus, Properties 1–3 and Properties 7–10 are satisfied. Property 4 is ensured by Line 40. What remains to be shown is that Property 6 is satisfied as well.

Note that $r$ is created such that $r.out.ID = ID$ and that by Property 3 of a valid relay parameter the relay $r'$ with $r'.ID = r.out.ID$ is valid: i.e., Property 6a) holds true.

According to Property 4 of a valid relay parameter and the fact that $r'.level \geq 0$ according to Property 3 of a valid relay parameter, Property 6b) of a valid relay holds as well.

For Property 6c) of a valid relay, first note that before the relay parameter was received, the corresponding message was stored in $r_s.Buf$ for a relay $r_s$ before and $r_s.out.ID = outID$ due to Property 12. By Property 1 of a valid relay parameter, $r_s$ was lingering-valid, thus $r_s.sinkRID = RID(outID)$ by Property 6b) of a valid relay. Thus, according to Property 5 of a valid relay parameter, $(key, \perp, r'') \in r'.In$ for some lingering-valid relay $r''$ and $r''.sinkRID = RID(outID)$. We now prove that $r''$ is also out-confirmed: When $m$ is received by $r$, for every $key' \in Key$ such that $(key', \perp, r''') \in r.In$ for some relay $r'''$ is replaced by $(key, RID(r_s), \perp)$. Since there was at least one key $key'$ feasible for $r_s$ in $Keys$ (which implies $key' \in r_s.out.Key$ by definition), afterwards $(key', RID(r_s), \perp)$ holds. This implies that $r_s$ is confirmed. Together with Property 13 of a valid relay parameter, we inductively obtain that $r''$ is out-confirmed as well. Since upon creation of the relay $r$, $key$ is put into $r.out.Key$, Property 6c) of a valid relay holds for $r$. □

**Lemma 6.57.** *If the application is deliberate, each relay parameter created from a valid relay $r$ by sending a message containing a reference of $r$ via a valid relay is a valid relay parameter. Furthermore, every valid relay parameter is either received by a sink and turned into a relay or remains valid.*

*Proof.* Note that every relay parameter is created with a unique key (which satisfies Property 2) and with $ID = r'.ID$ where $r'$ is the relay from which the relay parameter is created. Thus, if that relay $r'$ is valid, Property 3 holds. Since we assume the relay parameter is sent via a valid relay, Property 1 initially holds for every relay parameter, too. For Property 4 to Property 6, also check the way a

relay parameter is created. For Property 5, additionally check the fact that the relay $r''$ via which the relay parameter is sent is valid by the assumption that the application is deliberate and that it must be a non-sink relay (otherwise no relay parameter would be created). For Property 7 to Property 11, note that since $key$ is uniquely created, no such message or relay can exist at that point in time. Property 12 is also implied by the way a relay parameter is created and the fact that it is sent via a valid relay (which, in particular, satisfies Property 6c)). For Property 13 note that the sequence specified there is empty since the newly created message is initially put into $r''.Buf$.

We now check that every valid relay parameter $(key, ID, level, sinkRID)$ contained in a message $m$ in a buffer $r.Buf$ remains valid as long as $m$ has not been transmitted and processed. In the following, let $r'$ be the relay such that $r'.ID = ID$. As long as $r$ is not merged, Property 1 remains valid due to Lemma 6.53 and the fact that as long as $m \in r.Buf$, $r.Buf \neq \emptyset$. If $r$ is merged when Property 1 is fulfilled, then the resulting relay $r_n$ is valid according to 6.53 and the same applies to $r_n$ then. Property 2 again follows from the fact that relay parameters are created with a unique key. For Property 3, first of all note that relay parameters are never changed. In addition, note that according to Lemma 6.50 the relay $r'$ such that $r'.ID = ID$ remains valid unless it is deleted by the application or merged, which does not happen because according to Property 5, $r'.In \neq \emptyset$ and because the application is deliberate. The fact that relay parameters are never changed also implies Property 4. Let us now assume that Property 5 becomes false. Note that as long as $(key, \perp, r'') \in r'.In$, $r''$ remains being lingering. Thus Lemma 6.53 implies that $r''$ remains being lingering-valid. Note that the $sinkRID$ is never changed for a lingering-valid relay (due to Property 9 of a valid relay and the pseudocode). So in this case, Property 5 continues to hold. Therefore, assume that a triple $(key, \perp, r'')$ for some key $key$ and some lingering-valid relay $r''$ owned by the same node as $r'$ is removed from $r'.In$. According to the pseudocode, this happens at the following occasions:

1. In Line 21, in which case $r'$ receives a message using key $key$, which contradicts Property 8.

2. In Line 77, but this requires the existence of a PROBEFAIL$(key, keySequence)$ message for some $keySequence$ contradicting Property 9.

3. In Line 86, in which case prior to the removal of the triple from $r'.In$ a NOT-AUTHORIZED$(Keys, outID)$ message such that $r''.out.Key \subseteq Keys$ must have been in transit to $RL(r')$. Since $r''$ is lingering-valid, this contradicts Property 6c) for $r''$ together with Property 10 for the relay $r'''$ such that $r'''.ID = r''.out.ID$.

4. In Line 106, which is executed only if Property 3 is violated for $r'$, which would contradict Property 3.

5. In Line 143, which is executed only if Property 4 is violated for $r'$, which would contradict Property 3.
6. In Line 120, but that line is executed only if Property 8 of $r'$ is violated, which cannot be the case according to Property 3 of the relay parameter we consider.
7. In Line 125, but that line is executed only if $r''$ does not exist or is a sink relay, which contradicts Property 5.
8. In Line 127, but that line is executed only if $r'$ is dead, which contradicts $r'$ being valid.
9. In Line 164, but if that line is executed Property 9 must have been violated for $r''$, contradicting the fact that $r''$ is a lingering-valid relay.
10. In Line 184, but this requires an OUT-RELAY-CLOSED($r''.out.ID$) message to be received which contradicts the fact that $r''$ is lingering-valid and Property 1 for the relay $r'''$ such that $r''.out.ID = r'''.ID$.
11. When **delete** $r'$ is called: this does not occur, due to Lemma 6.40 (note that $r'$ is valid by Property 3), the fact that $r'.In \neq \emptyset$ and the assumption that the application is deliberate.
12. When $r''$ is merged into some new relay $r'''$. But then $(key, \bot, r'')$ is replaced by $(key, \bot, r''')$ and $r'''$ is lingering-valid as well according to Lemma 6.45.

For Property 6 note that the relay references inside a message are never changed. For Property 7 note that this could become violated only if $r'''$ is created from a relay parameter with key $key$. This, however, would contradict Property 2. For Property 8 note that whenever the key of a message is set, it is set to a key from $s.out.Key$ for a relay $s$, which cannot be $key$ due to Property 7. For Property 9 note that whenever an existing PROBEFAIL($key, keySequence$) message is received and thereby causes a new PROBEFAIL($key, keySequence'$) message, only $keySequence$ is truncated from the end to obtain $keySequence'$. Thus, the only occasion at which Property 9 might become false is when a PROBEFAIL($key, keySequence$) message is newly created (not as a result of a former such message). This only happens at a sink $s$ upon receipt of a PROBE($ControlKeys, keySequence$) message with $key \in ControlKeys$. However, this requires this PROBE() message to have been in $r'''.Buf$ for some relay $r'''$ with $r'''.out.ID = s.ID$, which contradicts Property 10 since $s.out.ID = \bot$. For Property 10 check that every newly created PROBE($ControlKeys, keySequence$) message does not violate this property, due to the way these messages are created and the fact that for $key \in ControlKeys$ they are sent via $r''$ only and $r''$ is lingering-valid. Thus, assume a PROBE() message $m_p$ is delivered from a buffer $r_i.Buf$ to the relay layer with RID $RID(r_i.out.ID)$. There, for the relay $r_{i+1}$ with $r_{i+1}.ID = r_i.out.ID$, $r_{i+1}.out.Key$ is appended to $keySequence$. If there is a message containing a relay parameter with a key contained in $ControlKeys$

(i.e., $m$ according to Property 2), this key is removed from $ControlKeys$. So either the resulting PROBE($ControlKeys'$, $keySequence'$) message $m'_p$ does not have $key \in ControlKeys'$ (and thus cannot contradict Property 10) or it otherwise also fulfills the requirements of Property 10. For Property 11 check both cases in which an IN-RELAY-CLOSED($Keys, RID, ID$) message with $key \in Keys$ is sent. The first is in Line 132, which requires a relay $r'''$ with $key \in r'''.out.Key$ contradicting Property 7. The second is in Line 167, which requires a PING($ID, level, sinkRID, key$) message. Property 5 for the valid relay parameter we consider implies that there is some lingering-valid relay $r''$ such that $(key, \bot, r'') \in r'.In$. Property 9 for the relay $r'$ then implies that there cannot be a PING($r'.ID, level, sinkRID, key$) message for any $level$ and any $sinkRID$. Since $r'.ID = ID$ (by definition of $r'$), a PING($ID, level, sinkRID, key$) message cannot have existed and thus Line 167 is not executed. Property 12 is implied by the fact that messages inside buffers are not changed (they are only removed once they have been transmitted and processed) and that $r$ is lingering-valid and Lemma 6.47. Last, for Property 13 notice that $r''$ is lingering-valid according to Property 5 and that every other relay in the sequence specified in Property 13 is lingering-valid as well (these are even valid according to the definition of $r''$ being lingering-valid, which implies that they are alive). Thus, all these relays remain lingering-valid according to Lemma 6.53 (note that they are replaced by the resulting relay when they are merged). Furthermore, note that for a fixed relay $r_i$ in the sequence, according to the pseudocode, only $RL(r_i)$ could send an IN-RELAY-CLOSED($Keys, RID(r_i), r_i.out.ID$) message with $key \in Keys$. However, according to the pseudocode, a message IN-RELAY-CLOSED($Keys, RID(r_i), r_i.out.ID$) with $key \in Keys$ can be sent only if $RL(r_i)$ receives a PING($r_i.out.ID, level, sinkRID, key$) message for some $level$ and $sinkRID$ and $key \notin r_i.out.Key$, so no such message is ever sent. Now check that a valid relay $r_{i+1}$ removes an element $(key, RID(r_i), \bot)$ from $r_{i+1}.In$ only if it receives an IN-RELAY-CLOSED($Keys, RID(r_i), r_i.out.ID$) message with $key \in Keys$, so the property continues to hold.

Next assume a message $m$ containing a valid relay reference is transmitted from $r.Buf$ to the node whose relay layer has the RID contained in $r.out.ID$ and let $s$ be the relay with $s.ID = r.out.ID$. Note that due to Property 12, this message must have a valid header for $s$. There are two options then: If $s$ is a sink, due to Property 6, the corresponding message is not discarded by the protocol executed when the message is received (see Line 36). Thus according to the pseudocode and the fact that Property 4, Property 6 and Property 7 hold, the relay parameter will be turned into a relay (see Lines 36–50). If $s$ is not a sink, i.e., $s.out.ID \neq \bot$, a message $m''$ that only differs from $m$ in its header is put into $s.Buf$ and only Property 1, Property 12 and Property 13 can be invalid for the new message if they held for the old one. For Property 1 note that since $r$ is lingering-valid, $s$ is valid due to Property 6a) for $r$ and lingering due to Property 6c) for $r$: i.e., $s$ is lingering-valid as well. For Property 12 note that according to the pseudocode, when $m''$ is created, it is created with header $(Keys, RID(s.ID), s.out.ID)$ where $Keys$ equals $s.out.Key$ at that point in time. Since $s$ is lingering-valid according to Property 1,

there is a feasible key in $s.out.Key$ at that point in time (due to Property 6c) for $s$) and thus $Keys$ contains a key that is feasible for $r$. For Property 13 first of all recall that Property 12 held for $m$ in $r.Buf$. Let $(Keys, r.ID, sr.out.ID)$ be the header of $m$. When $m$ is received by $s$, for every $key' \in Key$ such that $(key', \bot, r''') \in s.In$ for some relay $r'''$ is replaced by $(key, RID(r), \bot)$. Since there was at least one key $key'$ feasible for $r$ in $Key$ (which implies $key' \in r.out.Key$ by definition), afterwards $(key', RID(r), \bot)$ holds. Thus Property 13 also holds for the new sequence, which is the previous sequence appended by $s$ (note that the other relays in the sequence cannot cause the property to become untrue as argued before). □

We are now ready to prove the closure property, as formalized by Theorem 6.22, which we recap here:

**Theorem 6.22.** *If the application is deliberate, in every computation that starts in a legal state, every state is legal.*

*Proof.* The idea of the proof is the following: First, we argue that every valid relay that is not deleted by the application remains valid or is merged into a valid relay and that every dead-valid relay remains dead-valid unless it is removed completely. Second, we argue that every valid relay that is deleted by the application becomes a dead-valid relay. Third, we argue that every valid relay parameter remains valid unless it is received by a sink. Fourth, we prove that every new relay parameter is a valid relay parameter. Last, we show that every additionally created relay is a valid relay from the beginning. Altogether, this proves that the succeeding state of every legal state is legal as well.

The claim that that every valid relay that is not deleted by the application remains valid or is merged into a valid relay is directly implied by Lemma 6.50. The fact that every dead-valid relay remains dead-valid unless it is removed completely is stated by Lemma 6.54.

The second claim follows from Corollary 6.55. Note that the third claim follows from Lemma 6.57.

To see that every new relay reference is valid relay reference, first of all note that in a legal state, every new relay reference can only be created from a valid relay and only sent via a valid relay. To this end, Lemma 6.57 can be applied for every new relay parameter, yielding that the new relay parameter is valid.

To see that every additional relay is a valid relay from the beginning, first of all note that every sink relay that is newly created (via **new Relay**) is created as a valid relay by construction. Every non-sink relay newly created is created upon receipt of a relay parameter (which must be a valid relay parameter in a legal state). Lemma 6.56 gives that the resulting relay is valid then.

As argued before, this finishes the proof of the theorem. □

### 6.3.5. Additional Proofs for Theorem 6.23

In the last subsection of this section we prove Theorem 6.23, which basically implies the convergence property of self-stabilization. This, together with the previous results yields the completion of the self-stabilization proof. We begin with proving that every alive relay that fulfills certain conditions will eventually be deleted or become valid. After that, we stepwise prove that every of these conditions will eventually be fulfilled for every relay that is not removed completely. Putting this together, we obtain that every initially existing relay will eventually have become valid or gotten deleted. Note that we have already proven that valid relays remain valid. Thus, the next step is to show that all newly created relays are valid as well. Since we know that relays created from valid relay parameters become valid relays, we prove that every invalid relay parameter will cease to exist in finite time. Unfortunately this is not yet enough to prove that eventually there will be no invalid relay parameters. Even if we restrict the application to send references only via valid relays, the application could possibly send the reference of an invalid relay and thus cause new invalid relay parameters. The way out is to prove by induction on the level of the relays and to prove that for every $i$, all relays of level up to $i$ are eventually valid. This works because a new relay must have a higher level than the relay it was created from. So as soon as all relays up to level $\ell$ are valid, no new non-valid relays of level at most $\ell$ can emerge. In the end, when we restrict the application to not send the reference of a relay that exceeds a certain level at all, we obtain the desired claim of Theorem 6.23.

The first result states that a relay for which the following conditions are satisfied will eventually be valid:

**Lemma 6.58.** *If the application is deliberate, for every alive relay $r$ such that in some state $S$ (i) Properties 2–4 hold for $r$ or $r$ will be merged after $S$, (ii) every $r_s \in s(r)$ will not be deleted, and (iii) either $r.out.ID = \perp$ or Property 6 holds for every $s(r)$ in every state after $S$, there is a state $S'$ such that $cs_{S''}(r)$ is valid in every state $S'' \geq S'$.*

*Proof.* Assume that the application is deliberate. Consider an arbitrary alive relay $r$ such that in some state $S$ Properties 2–4 hold for $r$ or $r$ will be merged after $S$, every $r$ will not be deleted in finite time, $r_s \in s(r)$ will not be deleted, and either $r.out.ID = \perp$ or Property 6 holds for every $s(r)$ in every state after $S$.

First of all, note that $cs'_S(r).state = alive$ in every state $S'$ after $S$ by assumption. Note that as soon as Property 1 holds for some $cs(r)$, it holds as long as this relay exists (i.e., until it is merged) according to Lemma 6.48. Furthermore, note that when $r$ is merged, Property 1 holds for the resulting relay according to Lemma 6.45. Thus, all that needs to be proven is that if $r$ will not be merged, Property 1 eventually holds for $r$. According to the pseudocode, only $RL(r)$ could send an OUT-RELAY-CLOSED($r.ID$) message but does not do so when $r$ exists and is alive. Thus, as soon as all OUT-RELAY-CLOSED($r.ID$) messages initially in the system have been transmitted and processed, Property 1 holds for $r$.

Second, note that if $r$ will not be merged after $S$, Properties 2–4 hold for $r$ by assumption and continue to hold for $r$ by Lemma 6.37. Otherwise, according to Lemma 6.45, as soon as $r$ is merged, they hold for $cs(r)$ and every $r_s \in s(cs(r))$ according to Lemma 6.37 and Lemma 6.45.

Furthermore, if $r.out.ID = \perp$, Property 5 for $cs(r)$ will be satisfied after the next execution of TIMEOUT due to Line 98 and Line 109 and the fact that in case the node owning $cs(r)$ is inactive, $cs(r)$ would be removed completely upon TIMEOUT (see Line 137). This property will also be satisfied forever due to Lemma 6.37 and Lemma 6.45. Otherwise, Property 6 holds by assumption in $S$ and every subsequent state.

For the remaining properties, note that they hold after every merge according to Lemma 6.45. Furthermore, they continue to hold as soon as they hold according to Lemma 6.37 and Lemma 6.39. Thus, it is sufficient to show that they eventually hold for $r$ if $r$ will not be merged.

We consider Property 7. First of all, observe that whenever an element is added to $r.In$, this is either of the form $(key, RID, \perp)$ with $RID \neq \perp$ (see Line 22) or of the form $(key, \perp, r'')$ for some non-sink relay $r''$ owned by $RL(r)$. In addition, the TIMEOUT action in Line 125 ensures that after its first execution, $r.In$ only contains triples of the form $(key, RID, \perp)$ with $RID \neq \perp$ or $(key, \perp, r'')$ for some non-sink relay $r''$ owned by $RL(r)$. Last assume that for a triple $(key, \perp, r'') \in r.In$ such that $r''$ is a non-sink relay, $r''$ is completely removed. Check in the pseudocode all occasions at which a relay $r''$ is completely removed: except for three occasions, whenever a relay $r''$ is removed, all triples $(key, \perp, r'')$ are removed from $r'.In$ for every other relay $r'$. The first exception to this is in Line 129, in which case $r''$ is a sink relay. The second exception is in Line 134, in which case the check in Lines 130–131 ensures that no such triple $(key, \perp, r'')$ exists in $r.In$. The third exception is in Line 137, in which case due to the check in Lines 135–136 either $r''$ is a sink relay or $r.In = \emptyset$. Thus, Property 7 holds eventually and forever.

Note that Property 8 is satisfied by Line 120 and the fact that whenever a new triple is put into $cs(r).In$ either $key$ is uniquely created (see Line 9) or the first parameter $key$ already existed as the first parameter of another triple that is removed before (see Lines 21–22).

For Property 9 check that any PING() message with first parameter $r.ID$ is sent only by $RL(r)$ as PING$(r.ID, r.level, r.sinkRID, key)$ and only such that $(key, RID, \perp) \in r.In$ for some $RID$ (this happens in Line 122). If $r.out.ID = \perp$, then as soon as Property 5) holds, it will hold forever implying that $r.level$ and $r.sinkRID$ will never change. Otherwise, by assumption, Property 6 holds for $r$, which implies (in particular due to Property 6a) and Property 6b)) that $r.level$ and $r.sinkRID$ will never change. Thus as soon as $r.level$ and $r.sinkRID$ do not change any more and all PING() messages with first parameter $r.ID$ existing at this point in time have been transmitted and processed, Property 9 holds and holds forever.

To see that Property 10 will eventually become true, check in the pseudocode that a NOT-AUTHORIZED$(Keys, r.ID)$ message is sent only in Line 64 and only by $RL(r)$

and only if $RL(r)$ received a message $m$ with header $(Keys, senderRID, r.ID)$ and the message is sent only to the relay layer with RID $senderRID$. Furthermore, for this line to be executed, $m$ does not have a valid header for $r$: i.e., there is no $(key, senderRID, \bot) \in r.In$ and no $(key, \bot, r') \in r.In$ such that $r'.sinkRID = senderRID$. Thus a NOT-AUTHORIZED$(Keys, r.ID)$ message violating the requirements of Property 10 is never sent out and as soon as all of these messages initially in the system have been transmitted and processed, Property 10 will hold forever.

All in all, after some state, $r$ will be valid and remain valid forever. □

**Lemma 6.59.** *Every relay $r$ that does not satisfy Property 2, Property 3 or Property 4 will be removed completely in finite time or Properties 2–4 will eventually hold for $r$.*

*Proof.* Consider an arbitrary relay $r$. Assume that $r$ is not removed completely in finite time (otherwise, we are done).

If Property 2 is violated for $r$, Line 96 ensures that $r$ will be removed completely or a conflicting relay with the same ID will be removed completely such that the property holds afterwards. If Property 3 is violated for $r$, Line 107 ensures that $r$ will be removed completely.

We now prove that Property 4 will hold eventually and forever. Therefore, check that in case the property is violated for $r$, Line 144 of the TIMEOUT action makes sure it becomes satisfied (either by completely removing $r$, which would represent a contradiction, or by completely removing the other relay $r'$ such that $r'.out.Key \cap r.out.Key \neq \emptyset$). Furthermore, note that the only occasion at which a key $key$ is added to $r'''.out.Key$ for a relay $r'''$ owned by the same node as $r$ is in Line 45, in which due to Line 40 $key \notin r.out.Key$ holds. Thus, once Property 4 holds, it will hold forever. □

Note that the fact that sink relays cannot become merged, Lemma 6.58 and Lemma 6.59 directly imply the following corollary:

**Corollary 6.60.** *If the application is deliberate, for every alive sink relay $r$ that is not deleted there is a state $S$ such that $r$ is valid in every state $S' \geq S$.*

We now aim at a similar result for the non-sink relays (i.e., the relays $r$ such that $r.out.ID \neq \bot$).

**Lemma 6.61.** *If the application is deliberate, for every relay $r$ such that Property 6a) holds for $r$, $ls(r) = \times$ or there is a state $S$ such that Property 6 holds for $cs_{S'}(r)$ in every state $S' \geq S$.*

*Proof.* In the following, we assume that the application is deliberate. Consider an arbitrary relay $r$ such that Property 6a) holds for $r$ and assume that $ls(r) \neq \times$ (otherwise we are done). Note that Lemma 6.59, Lemma 6.45 and Lemma 6.37 imply that in some state $S_1$ and every subsequent state, Properties 2–4 will hold for $cs_{S_1}(r)$. In the following, let $r'$ be the relay such that $r'.ID = r.out.ID$.

First of all, Lemma 6.30 directly implies that there is a state $S_2$ after $S_1$ such that in every state $S_2' \geq S_2$ Property 6c) holds for $cs_{S_2'}(r)$ and that $r'$ will not be deleted or merged. Thus, according to Lemma 6.50 and the fact that $r''.out.ID$ is never changed for any existing relay $r''$, Property 6a) holds for $cs_{S'}(r)$ for every $S' \geq S_2$.

For Property 6b), note that Lemma 6.30 additionally implies that in every state $S' \geq S_2$ there will be a tuple $(key, RID(r), \bot) \in r'.In$ for at least one $key \in cs_{S'}(r).out.Key$. Then, during TIMEOUT, $RL(r')$ will send a PING$(r'.ID, r'.level, r'.sink, key)$ message to $RL(r)$ (see Line 122). Upon receipt of this message, $RL(r)$ will either completely remove $cs(r)$ (which would represent a contradiction) or update the values of $cs(r)$ such that Property 6b) is fulfilled for $cs(r)$ (note that $r'.level \geq 0$ since $r'$ is a valid relay). Let $S_3$ be the state such that this has happened already. Since Property 9 holds and continues to hold for $r'$, as $r.level$ and $r.sinkRID$ are changed only due to the receipt of a PING() message, and because of Lemma 6.45, Property 6b) holds for $cs_{S'}(r)$ in every $S' \geq S_3$.

All in all, we obtain the claim of the lemma. □

**Lemma 6.62.** *If the application is deliberate, for every alive relay $r$ such that Property 6a) holds for $r$, there is some $r_s \in s(r)$ that will be deleted or there is a state $S$ such that $cs_{S'}(r)$ is valid in every state $S' \geq S$.*

*Proof.* In the following, we assume that the application is deliberate. Consider an arbitrary alive relay $r$ such that Property 6a) holds for $r$ and assume that there is no $r_s \in s(r)$ that will be deleted (otherwise we are done). Note that there is a state $S_1$ such that Property 2, Property 3 and Property 4 hold for $cs_{S_1'}$ in every state $S_1' \geq S_1$ according to Lemma 6.37, Lemma 6.59 and Lemma 6.45. Furthermore, Lemma 6.61 implies that there is a state $S_2$ such that Property 6 holds for $cs_{S_2'}(r)$ in every state $S_2' \geq S_2$ (note that $ls(r) = \times$ would require some $r_s \in s(r)$ that would be deleted). Thus we can apply Lemma 6.58 yielding that there is a state $S_3$ such that $cs_{S_3'}(r)$ is valid in every state $S_3' \geq S_3$. □

**Lemma 6.63.** *If the application is deliberate, for every alive relay $r$ such that $r.out.ID \neq \bot$, there is some $r_s \in s(r)$ that will be deleted or there is a state $S$ such that $cs_{S'}(r)$ is valid in every state $S' \geq S$.*

*Proof.* Consider an arbitrary alive relay $r$ such that $r.out.ID \neq \bot$ and assume that there is no $r_s \in s(r)$ that will be deleted (otherwise we are done), which also implies that $ls(r) \neq \times$ (note that unless it is merged, an alive relay is never removed completely without being deleted first).

According to Lemma 6.34, there is a sequence of relays $(r_1 = r, r_2, r_3, \ldots, r_k)$ such that for all $i \in \{1, \ldots, k-1\}$, $r_{i+1}.ID = r_i.out.ID$, $r_k.out.ID = \bot$, and all relays $r_i$ for $i \in \{2, \ldots, k\}$ are and remain alive forever. We now prove the claim of the lemma via induction on $i$ (starting with $i = k$).

For the induction base, note that $cs(r_k) = r_k$ in every state because $r_k$ is a sink relay. As argued before, $r_k$ will not be deleted, which is why we can apply

Corollary 6.60. This yields that there is a state $S_k$ such that $cs_{S'_k}(r_k)$ will be valid in every state $S'_k \geq S_k$. Thus, for $cs(r_{k-1})$, Property 6a) holds in finite time and forever. Lemma 6.62 yields that there is a state $S_{k-1}$ such that $cs_{S'_{k-1}}(r_{k-1})$ is valid in every state $S'_{k-1} \geq S_{k-1}$ (again, recall that $r_{k-1}$ will not be deleted according to the aforementioned).

Now consider an arbitrary $i \in \{1, \ldots, k-2\}$ and assume there is a state $S_{i+1}$ such that $cs_{S'_{i+1}}(r_{i+1})$ is valid in every state $S'_{i+1} \geq S_{i+1}$. Again, we can apply Lemma 6.62 to obtain that there is a state $S_i$ such that $cs'_{S_i}(r_i)$ is valid in every state $S'_i \geq S_i$. This finishes the induction.

All in all, there is a state $S$ such that $cs_{S'}(r)$ will be valid every state $S' \geq S$ and the claim of the lemma is proven. □

Note that Corollary 6.60 and Lemma 6.63 imply the following corollary:

**Corollary 6.64.** *If the application is deliberate, for every alive relay $r$, there is some $r_s \in s(r)$ that will be deleted or there is a state $S$ such that $cs_{S'}(r)$ is valid in every state $S' \geq S$.*

**Lemma 6.65.** *For every relay parameter $(key, ID, level, sinkRID)$ contained in a message $m$ in a buffer $r.Buf$ of a relay $r$ in some state $S$ there is a state $S' > S$ such that $(key, ID, level, sinkRID)$ does not exist in $S'$.*

*Proof.* Consider an arbitrary relay parameter $(key, ID, level, sinkRID)$ contained in a message $m_1$ in a buffer $r_1.Buf$ of a relay $r$ in some state $S$. Let the header of $m_1$ be $(Keys, senderRID, outID)$. Note that $m_1$ will be received in some state by the relay layer whose ID is contained in $outID$. According to the pseudocode, there are three options then: First, if $(Keys, senderRID, outID)$ is not a valid header for the relay with ID $outID$ or such a relay does not exist, the message is discarded. Second, if there is a sink relay with ID $outID$ and $m$ has a valid header for that relay, the relay parameter is discarded (if the message contains relay parameters that belong to different RIDs) or "unpacked" into a relay (as described in Section 6.2.4) and no longer exists afterwards. Third, if $m$ has a valid header for a non-sink relay $r_2$, a new message $m_2$ containing $(key, ID, level, sinkRID)$ is put into $r_2.Buf$. We now argue that the third case cannot occur infinitely often for $(key, ID, level, sinkRID)$.

Let $(r_1, r_2, \ldots)$ be the resulting sequence of relays. Assume for contradiction that this sequence is infinite. Note that since $r.out.ID$ is never changed for any existing relay and since $r_i.out.ID = r_{i+1}.ID$ must hold for every $i \in \{1, 2, \ldots\}$, every relay in the above sequence must exist during $S$. In particular, this implies $cs(r_i) = \times$ or $cs(r_i) = r_i$ for every $i \in \{1, 2, \ldots\}$ in every state. Since the number of relays is finite, there must be a cycle $r_i, r_{i+1}, \ldots, r_j = r_i$ in the above sequence of relays. According to Lemma 6.31, one of the relays in the cycle will be removed completely, yielding that the relay parameter is eventually discarded. This completes the proof of the lemma. □

**Lemma 6.66.** *In every computation $C$, there is a state $S_0$ such that every alive relay $r$ such that $r.level = 0$ is valid.*

*Proof.* First of all, note that according to Lemma 6.50, every existing valid relay remains valid or is merged into some valid relay, or it is deleted by the application (in which case the relay is not alive afterwards). Second note that any new relay $r$ such that $r.level = 0$ must be a sink relay created via **new Relay**. Observe that due to the way the variables are set for a new relay sink relay created via **new Relay**, every new such relay is a valid relay by construction. Last, we see that every alive relay $r$ such that $r.level = 0$ will become a valid relay or be deleted in finite time according to Corollary 6.64. Thus, all in all, eventually every alive relay $r$ such that $r.level = 0$ is valid and the claim of the lemma follows. □

**Lemma 6.67.** *If the application is deliberate and does not send any reference via a relay that is not valid, the following holds for every fixed $l \in \mathbb{N}_0$: If there is a state $S_l$ such that in every state $S'_l \geq S_l$, every alive relay $r$ such that $r.level \leq l$ is valid, then there is a state $S_{l+1}$ such that in every state $S'_{l+1} \geq S_{l+1}$, every alive relay $r$ such that $r.level \leq l + 1$ is valid.*

*Proof.* Let $l \in \mathbb{N}_0$ be some fixed number. Consider an arbitrary computation $C$ such that the application is deliberate and does not send any reference via a relay that is not valid. Furthermore, assume there is a state $S_l$ such that in every state $S'_l \geq S_l$ every alive relay $r$ such that $r.level \leq l$ is valid. Recall in the following that according to Lemma 6.50, every existing valid relay remains valid or is merged into a valid relay, unless it is deleted by the application (in which case it is not alive anymore afterwards). We prove the claim in four steps: First, we show that every relay parameter $(key, ID, level, sinkRID)$ such that $level \leq l + 1$ created after $S_l$ is a valid relay parameter. Second, we show that every invalid relay parameter $(key, ID, level, sinkRID)$ such that $level \leq l + 1$ will vanish in finite time. These two insights imply that there is a state $S'_l$ such that in every state after $S'_l$ every relay parameter $(key, ID, level, sinkRID)$ such that $level \leq l + 1$ is valid. As a third step, we show that every new relay $r$ such that $r.level = l + 1$ created after $S'_l$ is a valid relay. Last, we show that every invalid relay $r$ such that $r.level = l + 1$ alive in $S'_l$ will be deleted or become valid in finite time. This implies that there is a state $S_{l+1}$ such that in every state $S'_{l+1} \geq S_{l+1}$ every alive relay $r$ such that $r.level \leq l + 1$ is valid.

For first step, note that whenever a relay parameter $(key, ID, level, sinkRID)$ is created, it is created from an alive relay $r$ such that $r.level = level - 1$. Thus, after $S_l$, the relay that every relay parameter $(key, ID, level, sinkRID)$ such that $level \leq l + 1$ is created from is valid and we can apply Lemma 6.57, which together with the assumption that the application does not send any reference via a relay that is not valid yields that $(key, ID, level, sinkRID)$ is valid.

For the second step, note that according to what we have just shown, every invalid relay parameter existing in the system must have been created before $S_l$. Every such relay parameter will vanish in finite time according to Lemma 6.65.

Thus, we obtain that there is a state $S'_l$ such that in every state after $S'_l$ every relay parameter $(key, ID, level, sinkRID)$ such that $level \leq l + 1$ is valid.

For the third step, consider an arbitrary relay $r$ such that $r.level = l + 1$ that is created after $S'_l$. According to the pseudocode, since $r.level > 0$, it can only be created due to the receipt of a relay parameter $(key, ID, level, sinkRID)$ for some $key$, $ID$ and $sinkRID$ and $level = r.level$. Since $r.level = l + 1$, this relay parameter must have been a valid one. Thus, Lemma 6.57 can be applied again, yielding that $r$ is a valid relay.

For the fourth step, the claim that every invalid relay $r$ such that $r.level = l + 1$ alive in $S'_l$ will be deleted or become valid in finite time is directly implied by Corollary 6.64. As argued before, this finishes the proof of the lemma. □

Note that Lemma 6.66 and Lemma 6.67 imply the following corollary:

**Corollary 6.68.** *If the application is deliberate and does not send any reference via a relay that is not valid, then for every fixed $l \in \mathbb{N}_0$, there is a state $S_l$ such that in every state $S'_l \geq S_l$, every alive relay $r$ such that $r.level \leq l$ is valid.*

We now prove Theorem 6.23, which we restate as follows:

**Theorem 6.23.** *If the application is deliberate and does not send any reference via a relay that is not valid and for some arbitrary but fixed $l \in \mathbb{N}$ does not send the reference of a relay $r$ such that $r.level \geq l$, every computation will reach a legal state.*

*Proof.* Consider an arbitrary computation $C$ such that the application is deliberate and does not send any reference via a relay that is not valid and for some arbitrary but fixed $l \in \mathbb{N}$ does not send the reference of a relay $r$ such that $r.level \geq l$. The main claim we need to show in this proof is that there is a state $S$ such that in every state $S' \geq S$, every alive relay $r$ is valid. Once we have proven this, we can apply Lemma 6.40 and Corollary 6.55 to obtain that after $S$, every relay that becomes a dead relay (i.e., a relay was an alive relay and is deleted) is either removed completely immediately or becomes dead-valid. Since we know from Lemma 6.54 that every dead-valid relay remains dead-valid unless it is removed completely, the last missing piece of the proof is that that every dead relay existing in $S$ that is not dead-valid will be removed in finite time. This, however, is given by Theorem 6.19. All in all, we obtain that $C$ reaches a legal state.

As described before, we now prove that there is a state $S$ such that in every state $S' \geq S$, every alive relay $r$ is valid. Corollary 6.68 states that there is a state $S_l$ such that in every state $S'_l \geq S_l$, every alive relay $r$ such that $r.level \leq l$ is valid. Note that whenever a relay parameter $(key, ID, level, sinkRID)$ is created, it is created from an alive relay $r$ such that $r.level = level - 1$. Thus, in the whole computation, no relay parameter $(key, ID, level, sinkRID)$ such that $level > l$ is created. This implies that after all relay parameters initially in the system have vanished (which they will do according to Lemma 6.65), no new relay $r$ such that $r.level > l$ can be created at all. Let $k$ be the maximum value of $r.level$ for any

existing relay $r$ (which is well-defined then). Then throughout $C$, for every relay $r$, $r.level \leq m$ for $m := \max l, k$. Thus, Corollary 6.68 yields that there is a state $S_m$ such that in every state $S'_m \geq S_m$, every alive relay $r$ is valid and the proof of the above claim is finished. □

### 6.3.6. Proof of Theorem 6.26

The proof of Theorem 6.26 consists of three parts that are represented by the following three lemmas:

**Lemma 6.69.** *In every legal state, for every relay $r$, if **has-incoming($\hat{r}$)** returns false, then there is no relay $r'$ with an edge $(r', r)$ in the relay graph.*

*Proof.* Consider an arbitrary legal state $S$ and an arbitrary relay $r$. Assume for contradiction that in $S$ **has-incoming($\hat{r}$)** returns false and there is a relay $r'$ with an edge $(r', r)$ in the relay graph. Since $S$ is a legal state $r'$ must be valid or dead-valid. In any case, Property 6 holds for $r'$. In particular, Property 6c) implies that $r.In \neq \emptyset$. This contradicts the assumption that **has-incoming($\hat{r}$)** returns false. □

**Lemma 6.70.** *For every relay $r$, $|r.In|$ does not increase as long as $r$'s reference is not sent in a message.*

*Proof.* Observe in the pseudocode that in all lines other than Line 10, in which the reference of a relay is sent in a message, whenever $r.In$ of some relay $r$ is modified, either only elements are removed from it or one element is added but another is removed beforehand. Thus, the claim follows. □

**Lemma 6.71.** *In every computation consisting of only legal states, for every relay $r$ for which there is a state $S$ such that $r$ is alive during $S$ and after $S$ there is no edge $(r', r)$ in the relay graph for any relay $r'$, the following holds: if $r$ is not deleted when $r.In \neq \emptyset$, there is a state $S' \geq S$ such that $r.In = \emptyset$.*

*Proof.* In the following, we assume that the application is deliberate. Consider an arbitrary computation consisting of only legal states and an arbitrary relay $r$ for which there is a state $S$ such that $r$ is alive during $S$ and after $S$ there is no edge $(r', r)$ in the relay graph for any relay $r'$. Furthermore, assume that $r$ is not deleted when $r.In \neq \emptyset$ (otherwise, there is nothing to be shown).

Note that the reference of $r$ cannot be sent in a message after $S$ as in this case, a new implicit edge $(r', r)$ would be formed in the relay graph for the relay $r'$ via which the reference of $r$ is sent, yielding a contradiction. Therefore, Lemma 6.70 implies that $|r.ID|$ is monotonically decreasing after $S$.

Observe in the pseudocode that in all lines other than Line 10, in which a message containing a reference of $r$ would be sent, only elements of the form $(key, RID, \perp)$ are added to $r.In$. All other elements will vanish over time according to Lemma 6.35. Note that this lemma assumes $ls(r) \neq \times$. But if $r$ has a successor, then $r.In = \emptyset$

must have held before $r$ was merged and the claim follows. Otherwise, if $r$ is removed completely, then it must have been deleted before, for a valid relay is never removed completely without being deleted first. In this case, $r.In = \emptyset$ held when $r$ was deleted by assumption and the claim follows. Thus, all that remains to be shown is that every element of the form $(key, RID, \bot)$ will vanish from $r.In$.

Note that during TIMEOUT, $RL(r)$ sends a PING$(r.ID, r.level, r.sinkRID, key)$ message to the relay layer with RID $RID$ for every $(key, RID, \bot) \in r.In$ (see Line 122). Every relay layer that receives this message does not have a relay $r'$ with $r'.out.ID = r.ID$ (because otherwise there would be an edge $(r', r)$ in the relay graph) and thus sends an IN-RELAY-CLOSED$(\{key\}, RID, r.ID)$ message back to $RL(r)$ (see Line 167). Upon receipt, $RL(r)$ removes $(key, RID, \bot)$ from $r.In$ (see Line 171).

All in all, $r.In$ is eventually empty and the claim follows. □

These three lemmas allow us to prove Theorem 6.26, which we restate here:

**Theorem 6.26.** *In every computation consisting only of legal states, the following holds for every alive relay $w$: Whenever **has-incoming($\hat{w}$)** returns false, then there is no relay $v$ with an edge $(v, w)$ in the relay graph. Moreover, if the computation has a suffix in which there is no relay $v$ that has an edge $(v, w)$ to $w$ in the relay graph and $w$ is not deleted as long as **has-incoming($\hat{w}$)** is true, there is a state $S$ such that **has-incoming($\hat{w}$)** returns false in $S$ and every subsequent state until $w$ is deleted.*

*Proof.* The first part of the claim follows directly from Lemma 6.69. For the second part, note that the fact that there is a state $S$ as defined in the claim follows from Lemma 6.71. The fact that **has-incoming($\hat{w}$)** continues to return false until $w$ is deleted follows from Lemma 6.70 and the fact that a message containing $w$'s reference would cause an implicit edge with endpoint $w$ in the relay graph. This finishes the proof. □

## 6.4. Universal Relay Primitives

We introduce three primitives for the manipulation of edges of a relay graph and show that they are universal: i.e., by using them it is possible to get from any arbitrary weakly connected valid relay graph consisting of alive relays only to any other weakly connected valid relay graph consisting of alive relays only involving the same set of nodes. The primitives we present are an adaptation of the node primitives by Koutsopoulos et al. [KSS17] defined in Chapter 2. Our proofs will rely on the universality of those primitives.

For convenience, in the following for two distinct nodes $u$ and $v$, we say a node $u$ has a *relay $r$ to* another node $v$ if $v$ is the sink node of $r$ and $u$ stores $\hat{r}$ in one of its variables or there is a message in transit to $u$ that will cause such a reference to be created upon receipt.


(a) Left: Initial situation for **Relay Introduction**. Right: After $u$ has sent $\hat{s}$ to $v$ (via $r$).


(b) Left: Initial situation for **Relay Fusion**. Right: After $u$ has executed **merge**$(\{r, r'\})$.


(c) Left: Initial situation for **Relay Reversal**. Right: After $u$ has sent $\hat{s}$ to $v$ (via $r$) and deleted $r$.

Figure 6.5.: Visualization of the primitives in $\mathcal{IFR}$.

The set $\mathcal{IFR}$ of relay primitives consists of the following primitives (examples of which are presented in Figure 6.5):

**Relay Introduction** Assume a node $u$ has a relay $r$ to a node $v$ and another relay $s$ to a node $w$. Then $u$ may send $\hat{s}$ to $v$ (via $r$).

**Relay Fusion** Assume a node $u$ has two different relays $r$ and $r'$ such that the return value of same-target$(\hat{r}, \hat{r}')$ is $true$. Then $u$ may merge the two relays.

**Relay Reversal** Assume a node $u$ has two different relays $r$ and $s$ such that has-incoming$(\hat{r}) = false$. Then $u$ may send $\hat{s}$ via $r$ and subsequently delete $r$.

We now show that these relay primitives maintain weak connectivity (Section 6.4.1), that they are sufficient to transform arbitrary graphs consisting of the same nodes into each other (Section 6.4.2) and that with their help it is possible to transform protocols for the standard interconnection model into protocols for the relay model (Section 6.4.3).

### 6.4.1. Maintenance of Weak Connectivity with $\mathcal{IFR}$

The first result concerning the relay primitives in $\mathcal{IFR}$ we just introduced is that, similar to the original primitives for nodes, they preserve weak connectivity of the valid relay graph. This is formalized by the following theorem:

**Theorem 6.72.** *$\mathcal{IFR}$ preserves weak connectivity: i.e., if any of the primitives is applied to a weakly connected valid relay graph $G$ consisting of alive relays only, then the resulting graph $G'$ is also weakly connected.*

*Proof.* First, note that Relay Introduction does not delete any relay, thus its application cannot harm the connectivity of the relay graph. Second, observe that Relay Fusion only merges redundant relays. Last, Relay Reversal preserves weak connectivity because although $u$ deletes a connection to the sink node $v$ of $r$, the message sent causes an edge from a relay owned by $v$ to $s$ (and thus there is an undirected path from $u$ to $v$), see Figure 6.5(c)). □

### 6.4.2. Universality of $\mathcal{IFR}$

We now want to prove that the three relay primitives in $\mathcal{IFR}$ are *universal*, meaning that arbitrary relay graphs can be transformed into arbitrary other relay graphs (as long as all relays are alive initially and in the end). This is, more precisely, stated by the following theorem:

**Theorem 6.73.** *The primitives in $\mathcal{IFR}$ are universal in a sense that one can get from any weakly connected valid relay graph $G = (V, E)$ consisting of alive relays only to any other weakly connected valid relay graph $G' = (V, E')$ consisting of alive relays only, where w.l.o.g. $E$ and $E'$ consist solely of explicit edges.*

The proof of Theorem 6.73 will use the universality of the (node) primitives Introduction, Delegation, Fusion, and Reversal (see Chapter 2). The idea is to *emulate* these primitives by the above relay primitives in order to reduce the universality of the relay primitives to the universality of the node primitives. Before we start with the proofs, we introduce a set of necessary definitions:

**Definition 6.74** (Simple Relay Graph)**.** *A* simple relay graph *is a valid relay graph $G = (P \cup R, E)$ such that all edges in $E$ are explicit and all relays in $R$ are direct relays and alive and that every sink relay in $R$ has exactly one incoming connection.*

**Definition 6.75** (Corresponding Process Graph)**.** *For a simple relay graph $G$, we define the* corresponding node graph *as the multigraph $\Lambda(G) = (P, E')$ whose vertices are the nodes only and whose edge set contains an edge $(u, v)$ with $u, v \in P$ for every edge $(u', v')$ in $G$ such that $u', v' \in R$ and $u'$ is owned by $u$ and $v'$ is owned by $v$.*

Note that there is a one-to-one relationship between a simple relay graph and its corresponding node graph: i.e., given a node graph $G_P$, there is a (except for isomorphism) unique relay graph $G_R$ with $\Lambda(G_R) = G_P$.

**Definition 6.76** (Emulation of node primitives)**.** *A set of relay primitives $RP$ emulates a node primitive $p$ if for every simple relay graph $G_R$ and every possible application of $p$ to $\Lambda(G_R)$, for each resulting node graph $G'_P$ there is a simple relay*

*graph $G'_R$ with $\Lambda(G'_R) = G'_P$ that can be obtained from $G_R$ by applying primitives from $RP$ only.*

Now that we have this definition, we can state the following lemma about the possibility to emulate the four node primitives by the relay primitives in $\mathcal{IFR}$.

**Lemma 6.77.** *$\mathcal{IFR}$ emulates each of the node primitives Introduction, Delegation, Fusion, and Reversal.*

*Proof.* The proof strategy is the same for every of the four primitives: Let $G_R$ be an arbitrary simple relay graph. Further, let $G_P = \Lambda(G_R)$. We will then consider an arbitrary application of the particular primitive to $G_P$ and denote the resulting graph by $G'_P$. After that, we show that by applying primitives from $\mathcal{IFR}$ to $G_R$, it is possible to obtain a graph $G'_R$ with $\Lambda(G'_R) = G'_P$. Thus, in the following we will use these variable names.

We start with the Introduction primitive. Applying the Introduction primitive means that for some node $u$ with references of two other nodes $v$ and $w$ in $G_P$, $u$ sends a message to $v$ containing a reference of $w$ and keeps the reference. Thus, in the resulting graph $G'_P$, there is an additional edge $(v, w)$. In $G_R$, let $u$ send its relay to $v$ to $w$, which resembles a Relay Introduction. Subsequently, let $w$ create a new relay, send this via the received relay and then close the received relay. This then resembles a Relay Reversal. In the resulting relay graph $G'_R$, all that has changed in comparison to $G_R$ is that $v$ now has an additional direct relay to $w$. Thus, in $\Lambda(G'_R)$ all that has changed in comparison to $\Lambda(G_R)$ is that there is an additional edge $(v, w)$, thus this graph is isomorphic to $G'_P$.

Next, we deal with the Delegation primitive. Applying the Delegation primitive means that for some node $u$ with references to two nodes $v$ and $w$ in $G_P$ such that $u$, $v$ and $w$ are all different, $u$ sends a message to $v$ containing a reference a reference of $w$ and deletes the reference of $w$. Thus, the resulting graph $G'_P$ differs from $G_P$ in that there is an additional edge $(v, w)$ and the edge $(u, w)$ is removed. In $G_R$, let $u$ send its relay to $v$ to $w$ and delete the relay to $w$. This resembles a Relay Reversal. After that, let $w$ create a new relay, send this via the received relay, and close the received relay. Then this resembles a Relay Reversal, again. In the resulting relay graph $G'_R$, all that has changed in comparison to $G_R$ is that $v$ now has an additional direct relay to $w$ and that $u$ no longer has its relay to $w$. Thus $\Lambda(G'_R) = G'_P$, again.

For the Fusion primitive, it is obvious that Relay Fusion emulates the node primitive Fusion.

Last, applying the Reversal primitive means that some node $u$ that has a reference of some other node $v$ sends a reference of itself to $v$ and deletes its reference of $v$. In the relay graph, $u$ would create a new relay, send it via the relay to $v$ and subsequently delete its relay to $v$, which resembles a Relay Reversal. This finishes the proof. □

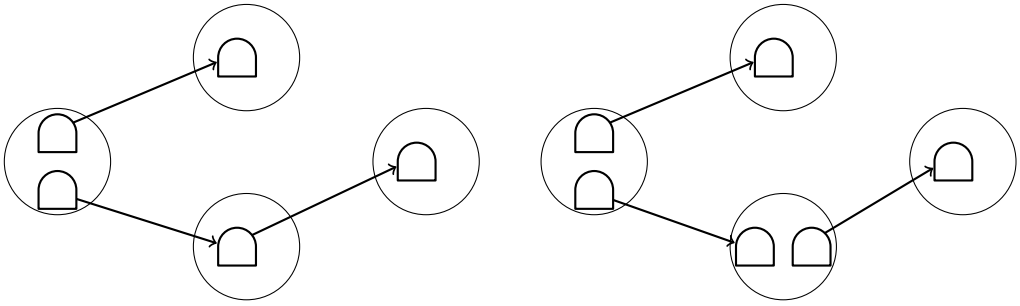
Applying Lemma 6.77 we can finally prove Theorem 6.73, which we recap as follows:

**Theorem 6.73.** *The primitives in $\mathcal{IFR}$ are universal in a sense that one can get from any weakly connected valid relay graph $G = (V, E)$ consisting of alive relays only to any other weakly connected valid relay graph $G' = (V, E')$ consisting of alive relays only, where w.l.o.g. $E$ and $E'$ consist solely of explicit edges.*

*Proof.* The idea of the proof is the following: Consider an arbitrary relay graph $G = (R \cup P, E)$ consisting of alive relays only and assume that $G$ is weakly connected. First, we show how to transform $G$ into a simple relay graph $G_1$ over the same nodes that is weakly connected as well. The universality of the node primitives from [KSS17] and Lemma 6.77 imply that it is possible to transform this graph into another simple relay graph $G_2$ with $\Lambda(G_2) = (P, E_2)$ by using the primitives in $\mathcal{IFR}$, which is defined such that $(w, v) \in E_2$ if and only if in $G'$ there is an edge $(r, s)$ such that $r$ is stored by $v$ and $s$ is stored by $w$. Last, we show how to transform $G_2$ into a graph isomorphic to $G'$, which finishes the proof. An example of the graphs used here is shown in Figure 6.6.

To transform $G$ into $G_1$, we proceed as follows. As long as there is still an indirect relay $r$ stored by any node $v$ with has-incoming$(\hat{r}) = false$, $v$ applies Relay Reversal as follows: $v$ creates a new relay $r'$, sends this relay via $r$ and subsequently closes $r$. This strictly decreases the number of indirect relays in every iteration. Note that as soon as there is no indirect relay $r$ with has-incoming$(\hat{r}) = false$ anymore, there cannot be any indirect relay at all (recall that there cannot be any cycles in the sequences of relay connections). Note that by Theorem 6.72, since we applied Relay Reversal only, the resulting graph $G_1$ is still weakly connected. Furthermore, $G_1$ is a simple relay graph. Thus, as described above, it is possible to transform this graph into a graph $G_2$ as described above.

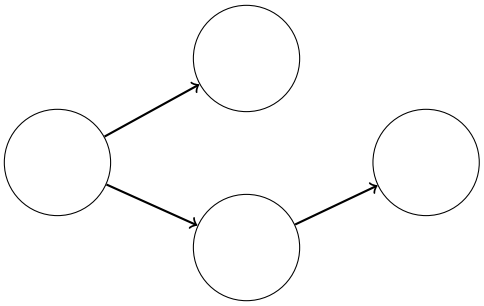
To transform $G_2$ into $G'$, consider an arbitrary sink relay $s$ in $G'$ and let $T$ be the subgraph of $G'$ that contains all relays $r$ with sink relay $s$. Note that $T$ is a tree. Thus, for an arbitrary relay $r$ in $T$, define $children_T(r)$ as the set of relays $q$ with an edge $(q, r)$ in $T$. Similarly, for an arbitrary relay $r \neq s$ in $T$, define $parent_T(r)$ as the relay $q$ for which there is an edge $(r, q)$ in $T$. Denote by $L_T(i)$ the set of relays at level $i$ of $T$. An example of these notions can be seen in Figure 6.7. By the definition of $G_2$, every node storing a relay $r$ of $T$ (in $G'$) stores (in $G_2$) a direct relay to each node storing a relay $r' \in children_T(r)$ (in $G'$). First of all, the node storing $s$ (the root of $T$) in $G'$ creates a new relay $s'$ (which in the end will be the equivalent of $s$). Then, it sends $s'$ to each node storing a relay $r \in children_T(s')$ (in $G'$) and closes each of the relays to a node storing a relay $r \in children_T(s')$ (in $G'$), i.e., the relays via which the relay was sent, thus performing a Relay Reversal. This way, every node storing a relay $r$ in $L_T(1)$ (in $G'$) receives a relay $r'$ whose endpoint is equivalent to $parent_T(r')$. Then, for every ascending level $i \geq 1$, every node storing a relay $r$ in $L_T(i)$ sends the relay it received via a relay of $T$ to each of the nodes storing relays in $children_T(r)$ and closes the relays via which it sent this relay, thus performing a Relay Reversal, too. Similarly, every node storing a relay in $L_T(i+1)$ receives a relay $r$ whose endpoint is equivalent to $parent_T(r')$. In the end, we obtain the desired tree $T$. Since $s$ and thus also $T$ was chosen arbitrarily



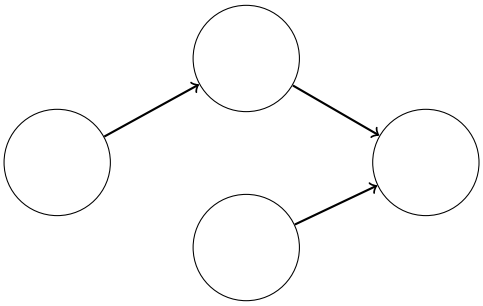
(a) Initial graph $G$.



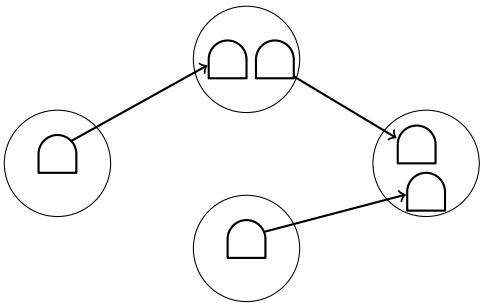
(b) Intermediate graph $G_1$.



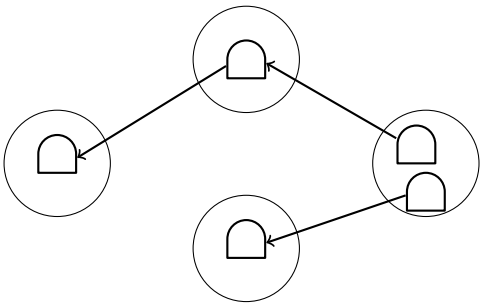
(c) Graph $CGP(G_1)$.



(d) Graph $CPG(G_2)$.



(e) Intermediate graph $G_2$.



(f) Final graph $G'$.

Figure 6.6.: Graphs used in the proof of Theorem 6.73. $G$ is supposed to be transformed into $G'$.

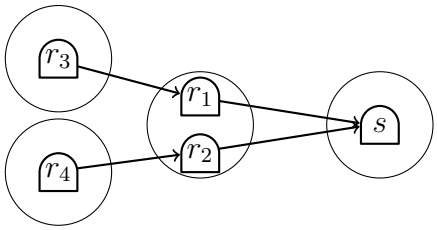



Figure 6.7.: Example of some notions introduced in the proof of Theorem 6.73. The depicted relay tree $T$ is rooted at $s$, $children(s) = \{r_1, r_2\}$, $parent(r_2) = s$ and $L_T(2) = \{r_3, r_4\}$.

and since there is no edge in $G_2$ that is not removed in the transformation, this finishes the proof. □

### 6.4.3. Using $\mathcal{IFR}$ to Adapt Classical Protocols to the Relay Model

One of the benefits of the relay model is that a wide range of protocols designed for the standard interconnection model (e.g., [Gal+14; Jac+14; Jac+12; ORS07; SR05]) can be adapted to the relay model. In [SSS16] it was shown that a wide range of protocols for static strongly-connected topologies that preserve weak connectivity can be transformed such that the interaction between nodes can be decomposed into the primitives Introduction, Delegation, and Fusion (Theorem 1 of that work). Lemma 6.77 of this chapter yields that these primitives can be emulated by the relay primitives in $\mathcal{IFR}$ such that the resulting graph consists of direct relays only. Putting this together, the aforementioned class of protocols can be adapted to the relay model.

## 6.5. Solving the $\mathcal{FDP}$ with Relays

We now describe a general framework to solve the finite departure problem in the relay model. With *general* we mean that our framework can be applied to a wide range of protocols that self-stabilize to a topology. First, we formalize some prerequisites in Section 6.5.1. After that, we describe in Section 6.5.2 the general idea of our framework. A more detailed description and the pseudocode of our protocol are then given in Section 6.5.3. Last, in Section 6.5.4 we prove that the protocol solves the finite departure problem and eventually does not affect the execution of the original protocol.

### 6.5.1. Prerequisites

To build the desired framework, we formalize the operations and treatments of variables of protocols that stabilize a topology in the relay model. We assume that the protocol that is being transformed, also called the *original protocol* in the following, satisfies the following three assumptions:

**Single-sink assumption** Every node $u$ owns exactly one relay that eventually is the only sink relay owned by $u$ and the only relay owned by $u$ with incoming connections, and this relay is denoted by $u.in$.

**Self-introduction assumption** Every node $u$ during every execution of Timeout sends $u.in$ via every direct relay stored by $u$.

**Direct relay assumption** The target topology does not contain indirect relays, and indirect relays are reversed whenever they exist (i.e., during Timeout if their reference is stored in the variable of a node or before the end of an action if they are created due to a message calling that action). Furthermore, references of indirect relays are never sent.

We argue that these assumptions are natural for protocols for self-stabilizing topologies and can easily be fulfilled by existing protocols adapted from the common interconnection model (see Section 6.4.3). The single-sink assumption is not a restriction. Regarding the self-introduction assumption, note that in general self-introduction is necessary for convergence, as otherwise nodes with no incoming connections would not be integrated into the network. As to the direct relay assumption, note that for every relay $r$ created due to the receipt of a message, has-incoming($\hat{r}$) = $false$ by definition. Thus, directly deleting $r$ is always possible. Furthermore, any protocol that would store or send the reference of an indirect relay could be transformed such that it first replaces the indirect relay $r$ by a direct relay $r$ to the same sink node whose reference it can then send. Thus, the third assumption does not represent a limitation either.

Our framework will introduce new actions that, without loss of generality, we assume to be distinct from the existing actions of the original protocol $P$ (the latter we refer to as the *protocol actions*). An exception to this is the TIMEOUT action of our protocol, which may contain the TIMEOUT action of $P$ as a subroutine. Furthermore, for each protocol action $A$ we define a REPLACE_A() action that is run instead of $A$ and may include a call of $A$ (denoted by $A()$) or not (in the latter case, the original action $A$ is never executed). We refer to those parameters of $A$ that are relay references by $r_1, \ldots, r_k$ (and assume that all additional parameters will be implicitly attached to the call of $A$ where that occurs). Since nodes do not have a direct access to the relays, but only deal with references of relays, we will omit the distinction between relays and their references in the following. We also introduce the following notion: When in the following we say that a node $u$ *performs a reversal on a relay* $r$ owned by $u$, this means $u$ sends $u.in$ via $r$ contained in a message of an arbitrary type specified by the original protocol (if there is no message with only one parameter, the additional parameters may be filled arbitrarily except for with relays other than $r$) and deletes $r$ right after that. This resembles the relay reversal primitive and is needed to have a general procedure to get rid of indirect relays.

### 6.5.2. Intuitive Approach Description

In this subsection we describe the major ideas and ingredients of our protocol.

First of all, in order to get rid of indirect relays, every node periodically applies a relay reversal on every indirect relay that has no incoming connection. Because of the direct relay assumption (which implies that the protocol will not permanently keep indirect relays), this procedure eventually causes all indirect relays to be gone.

Second, each leaving node $u$ will determine a staying node, called $u$'s *anchor*, which, once found, $u$ keeps an outgoing connection to until it leaves the system. This anchor will be used for the reversal of relays that $u$ wants to get rid of.

Third, each leaving node will attempt to get rid of all other non-sink relays (i.e., all but the one to the anchor). This can be achieved easily via a relay reversal. As soon as the non-sink relay $r$ in question does not have any incoming connections

anymore (which will be the case at some point according to the first part), the leaving node sends the reference of its relay to the anchor via $r$ and subsequently deletes the respective non-sink relay.

Fourth, each leaving node will try to get rid of all incoming connections. Note that, according to the above, eventually no leaving node will have incoming connections from a leaving node. Staying nodes with a connection to a leaving node $u$, however, periodically introduce themselves to $u$ according to the self-introduction assumption. Such a node $u$ can use this to ask the respective staying nodes to reverse their connections to $u$. Of course, this requires $u$ to send out references of its sink relays for this purpose, but we ensure that the corresponding actions called at the receiving nodes cause these relays (and the ones to be closed if they still exist) to be closed immediately. Aside from this exception, every leaving node does not accept new connections to it.

All in all, leaving nodes will eventually have no relays with incoming connections and only a single relay with an outgoing connection (to the anchor) and will thus be able to safely leave the system.

### 6.5.3. Detailed Description and Pseudocode

In this section, we give a brief summary of the purpose of the variables and actions used in our protocol. The full pseudocode is then presented in Listing 6.10 (for staying nodes) and Listing 6.11 (for leaving nodes).

Each node $u$ maintains the following variables:

**u.in** Reference to the unique sink relay of $u$.

**u.N** Pseudo-variable that serves as a union of all variables of $u$ that store relays in the original protocol (except for $u.in$). In our protocol, the only write access to this set will be to remove elements from it (meaning that the original protocol variable is set to an undefined value, $\perp$).

**u.D** Set of relays that $u$ wants to get rid of. Relays are added to this set if they still have incoming connections when they are supposed to be reversed. $u$ regularly checks whether there is a relay in $u.D$ that has no incoming connection anymore and, if so, performs a relay reversal on that relay.

**u.a-out** Non-sink relay to the anchor of $u$ (purposefully, only leaving nodes have a value $\neq \perp$ for this variable).

**u.a-in** Sink relay used to check whether the anchor is leaving or not (purposefully, only leaving nodes have a value $\neq \perp$ for this variable). As we will explain later, $u.In$ cannot be used for this purpose.

Of course, there may be additional variables from the original protocol, which we do not consider here. Furthermore, we assume all of the above variable names (except for $u.in$) to be distinct from the original protocol variable names (which can easily be achieved by renaming).

We continue by explaining the purpose of the various actions for staying and leaving nodes (and, correspondingly, the message types used for our framework). The full pseudocode of the protocol is then provided in Listing 6.10 (for staying nodes) and Listing 6.11 (for leaving nodes). The actions (and message types) and their purposes are:

**Timeout** At the beginning of TIMEOUT, a set of invariants for the variables is established (displayed in the comments in Lines 3–16 and Lines 44–60). The protocol will maintain these invariants: i.e., these lines only have an effect during the first run of TIMEOUT. After that, to get rid of these relays the executing node performs a reversal on every relay stored in $D$ that has no incoming connection (Lines 18–20 and Lines 62–65). Note that for leaving nodes this is done by sending an ASK-TO-REVERSE() message with parameter *in*. After that, leaving nodes will perform **stop** if the necessary conditions are fulfilled (see Lines 66–67). If this is not the case and if they have an anchor, they send one or two messages (depending on the value of has-incoming(*a-in*)) to the anchor, whose purpose is described below.

**Replace_A()** Staying nodes simply normally execute action $A$. Leaving nodes do not participate in the original protocol (as they are leaving anyway) and thus instead perform a reversal on all parameters of this action by sending an ASK-TO-REVERSE(*in*) message (in order to get rid of these references without harming weak connectivity).

**Ask-to-reverse(*out*)** An ASK-TO-REVERSE() message is only sent by leaving nodes. It indicates to the receiving node $u$ that it should not store any relay with a connection to relay *out*. Upon receipt, $u$ removes all references of relays to which this applies from $N$ or from *a-out* (they remain stored in $D$ to keep weak connectivity, but this is only temporary because, as mentioned before, the protocol eventually gets rid of the elements in $D$). A staying node will reverse *out* with a REVERSE(*in*) message to *out*. The behavior of a leaving node depends on whether it has an anchor (i.e., on whether *a-out* $\neq \perp$). If so, it will try to help the sink node of *out* to find an anchor by sending *a-out* to *out* in a REVERSE() message. Otherwise, it will just send an ASK-TO-REVERSE(*in*) message to *out*, thereby telling the sink relay of *out* that it is leaving. This may cause a message cycle which is, however, interrupted as soon as one of the two nodes gets an anchor.

**Ask-to-reverse-anchor(*out*)** Simply put, a leaving node $u$ sends an ASK-TO-REVERSE-ANCHOR($u.in$) message to ask its anchor whether it is leaving or staying. The answer is given by the response: Staying nodes respond with a REVERSE() message to indicate that the anchor is staying and leaving nodes respond with an ASK-TO-REVERSE() message to indicate that the anchor should be removed.

**Notify-anchor()** The NOTIFY-ANCHOR() action is necessary to handle a special case that may arise from initial states. It has no effect on staying nodes. Leaving nodes regularly send this message to their anchor during TIMEOUT. We will explain the full purpose of this action below.

**Reverse(*out*)** In general, this message type is used to introduce a staying node to a leaving node such that the latter can obtain an anchor. To fulfill this purpose, the protocol sends this message only with a parameter whose sink is staying (see Line 30 and Line 34) or believed to be staying (as it is the anchor of the sending node, see Line 89).

The reason why during TIMEOUT a leaving node $u$ sends an ASK-TO-REVERSE-ANCHOR($a$-$in$) message to their anchor only if $u.a$-$in$ has no incoming connections is as follows: If $u$ did not check that has-incoming($u.a$-$in$) is false, $u$ could continuously increase the number of messages containing a relay to $u.a$-$in$, which could prohibit $u$ from leaving (even if all other conditions were fulfilled). This is also why the variable $u.a$-$in$ is necessary and not simply $u.in$ may be used. There may be other connections to $in$ that would only vanish after $u$ has found an anchor that is staying. Thus has-incoming($u.in$) $= false$ may never be reached before $u$ has a staying anchor. On the other hand, ASK-TO-REVERSE-ANCHOR() is exactly sent to determine whether the supposed anchor is leaving. Without this message being sent, $u$ might continue to store a leaving node in $u.a$-$out$.

We also left open why every leaving node $u$ sends a NOTIFY-ANCHOR() message to its anchor during TIMEOUT if $u$ has one: i.e., to the sink node $v$ of $a$-$out$ if $a$-$out \neq \perp$ (c.f. Line 71). This is necessary because of the following special case: Due to the initial system state, it could be that $v$ is leaving and the sink of $u.a$-$out$ is $v.a$-$in$. Even worse, $v.a$-$out$ may have sink $u.a$-$in$, in which case it could happen that none of the has-incoming($u.a$-$in$) and has-incoming($v.a$-$in$) would ever become false (recall that the ASK-TO-REVERSE-ANCHOR() message is only sent if has-incoming($a$-$in$) $= false$). Thus, the NOTIFY-ANCHOR() message sent by $u$ informs $v$ of this situation such that it can renew its $a$-$in$ variable (c.f. Lines 97–99).

To simplify the pseudocode, we assume that the cleanup procedure of TIMEOUT (the lines that deal with initial states) is executed before the execution of each action. Furthermore, we assume that every message for which a relay parameter $out$ is specified in the pseudocode is ignored if $out$ is not a relay parameter.

Listing 6.10: Pseudocode for **staying** nodes

```
1  TIMEOUT
2    // Lines 3-16 deal with initial states
3    for all r ∈ getRelays such that r ∉ N ∪ D ∪ {in, a-out, a-in} do
4    |  D := D ∪ {r}
5    if not is-sink(in) then // make sure in is a sink relay
6    |  D := D ∪ {in}
7    |  in := new Relay
8    for all r ∈ N such that direct(r) = false // make sure all relays in N are direct
9    |  N := N \ {r}
```

```
10    |  D := D ∪ {r}
11    if a-in ≠ ⊥ then // make sure a-in is undefined
12    |  D := D ∪ {a-in}
13    |  a-in := ⊥
14    if a-out ≠ ⊥ then // make sure a-out is undefined
15    |  D := D ∪ {a-out}
16    |  a-out := ⊥
17
18    for all r ∈ D such that incoming(r) = 0 do
19    |  D := D \ {r}
20    |  # perform a reversal on r (thereby deleting r)
21    # perform TIMEOUT of the original protocol
22
23  REPLACE_A(r_1, ..., r_k)
24    A(r_1, ..., r_k) // execute action normally
25
26  ASK-TO-REVERSE(out)
27    for all v ∈ N such that same-target(out, v) do
28    |  N := N \ {v}
29    |  D := D ∪ {v}
30    SEND(out, REVERSE(in))
31    delete out
32
33  ASK-TO-REVERSE-ANCHOR(out)
34    SEND(out, REVERSE(in))
35    delete out
36
37  NOTIFY-ANCHOR()
38    // do nothing
39
40  REVERSE(out)
41    # perform a reversal on out (thereby deleting out)
```

Listing 6.11: Pseudocode for **leaving** nodes

```
42  TIMEOUT
43    // Lines 44-60 deal with initial states
44    for all r ∈ getRelays such that r ∉ N ∪ D ∪ {in, a-out, a-in} do
45    |  D := D ∪ {r}
46    if not is-sink(in) then // make sure in is a sink relay
47    |  D := D ∪ {in}
48    |  in := New Relay()
49    for all r ∈ N do  // make sure N is empty
50    |  N := N \ {r}
51    |  D := D ∪ {r}
52    if not is-sink(a-in) then // make sure a-in is a sink relay
53    |  D := D ∪ {a-in}
54    |  a-in := new Relay
55    if a-in = ⊥ then // make sure a-in exists
56    |  a-in = New Relay
57    if a-out ≠ ⊥ then // make sure direct(a-out) and has-incoming(a-out) = false if exists
58    |  if has-incoming(a-out) > true or direct(a-out) = false or is-sink(a-out) = true then
59    |  |  D := D ∪ {a-out}
60    |  |  a-out := ⊥
```

```
61
62      for all r ∈ D such that incoming(r) = 0 do
63      |  Send(r, Ask-to-reverse(in))
64      |  D := D \ {r}
65      |  delete r
66      if D = ∅ and has-incoming(in) = false and has-incoming(a-in) = false then
67      |  |  stop // since the only connection is to a-out, it is safe to leave
68      if a-out ≠ ⊥ then
69      |  if has-incoming(a-in) = false then
70      |  |  Send(a-out, Ask-to-reverse-anchor(a-in))
71      |  Send(a-out, Notify-anchor())
72
73  Replace_A(r_1, ..., r_k)
74      for all i ∈ {1, ..., k} do
75      |  let r be the relay via which the message calling this action was received
76      |  Send(r_i, Ask-to-reverse(r))
77      |  delete r_i
78
79  Ask-to-reverse(out)
80      if a-out = ⊥ then // ask out for its anchor
81      |  Send(out, Ask-to-reverse(in))
82      |  delete out
83      else
84      |  if same-target(out, a-out) then // anchor wants to leave
85      |  |  r' := {merge{out, a-out}}
86      |  |  a-out := ⊥
87      |  |  Send(r', Ask-to-reverse(in))
88      |  else // let out know the anchor
89      |  |  Send(out, Reverse(a-out))
90      |  |  delete out
91
92  Ask-to-reverse-anchor(out)
93      let r be the relay via which the message calling this action was received
94      Send(out, Ask-to-reverse(r))
95      delete out
96
97  Notify-anchor()
98      D := D ∪ {a-in}
99      a-in := new Relay
100
101 Reverse(out)
102     if a-out = ⊥ then
103     |  if direct(out) then // add out directly
104     |  |  a-out := out
105     |  else // ask out to send a direct relay
106     |  |  Send(out, Ask-to-reverse(in))
107     |  |  delete out
108     else
109     |  if same-target(out, a-out) then
110     |  |  a-out := merge{out, a-out}
111     |  else // reverse out with the help of the anchor
112     |  |  Send(out, Ask-to-reverse(a-out))
113     |  |  delete out
```

### 6.5.4. Analysis

To show that our protocol represents a self-stabilizing solution to the $\mathcal{FDP}$, we have to prove that the protocol never disconnects the set of active nodes, that all leaving nodes eventually become inactive and that all staying nodes remain active. In addition, we show that our transformation approach does not affect the behavior of the original protocol as soon as all leaving nodes are inactive. These goals are formalized by the following three theorems:

**Theorem 6.78.** *Let $G_a(S)$ be the relay graph induced by the set of* active *nodes and the relays owned by these nodes in state $S$. If in every computation $C$ of the original protocol that starts from a weakly-connected graph of active nodes, $G_a(S)$ remains weakly connected in every state $S \in C$, then the same holds for every computation of the transformed protocol.*

**Theorem 6.79.** *In every computation of the transformed protocol in which there is at least one staying node, every leaving node will eventually be inactive and every staying node will always remain active.*

**Theorem 6.80.** *In every computation $C$ of the transformed protocol, there is a state $S$ such that every leaving node is inactive, every staying node is active and there is a computation $C'$ of the original protocol starting from $S$ such that $C' = SUFFIX_C(S)$.*

The rest of this section consists in proving the correctness of these three theorems.

#### Proof of Theorem 6.78 - Connectivity

We first prove that the connectivity of the active nodes is not harmed by our framework, which is stated by Theorem 6.78. This theorem is restated in the following:

**Theorem 6.78.** *Let $G_a(S)$ be the relay graph induced by the set of* active *nodes and the relays owned by these nodes in state $S$. If in every computation $C$ of the original protocol that starts from a weakly-connected graph of active nodes, $G_a(S)$ remains weakly connected in every state $S \in C$, then the same holds for every computation of the transformed protocol.*

*Proof.* The main idea of the proof of this theorem is that a node executes **stop** only if it has at most one outgoing relay connection and no incoming relay connection. This way, the becoming inactive of a node does not harm the connectivity of the remaining graph. In addition, we show that the **delete** command is used only during the application of one of the relay primitives from $\mathcal{IFR}$, which preserve weak connectivity (see Section 6.4). Therefore, an execution of this command cannot disconnect the graph of active nodes either.

Note that the only occasion at which an active node $u$ stops is during TIMEOUT in Line 67 and that this line is only executed if $u$ is leaving. In this case, Lines 44–45 have been executed before, yielding that all relays are stored in one of the variables of $u$. Furthermore, Lines 49–51 have been executed, yielding $N = \emptyset$. Moreover, Lines 46–48 and Lines 52–54 have been executed, which ensures that *in* and *a-in* are sink relays. Additionally, due to the conditions of Line 66, $u$ has no relay with an incoming connection (otherwise the has-incoming commands would not return false). Altogether, $u$ is connected with at most one node (via $u$.*a-out*), thus the stopping of $u$ will not disconnect the node graph.

Observe in the pseudocode that whenever a node $u$ deletes a relay $r$, then either $r$ is a sink relay or, prior to deleting $r$, $u$ sends a message via $r$ that contains as one parameter the reference of a relay $r'$ stored by $u$ (i.e., $u$ applies the relay reversal primitive). This way, there is still a (not necessarily directed) path in the relay graph from $u$ to the sink node of $r$. Thus, as long as none of the actions of the original protocol disconnect the graph, the same holds for the transformed protocol. □

**Proof of Theorem 6.79 - Finite Departure**

The proof of Theorem 6.79, which basically states that the transformed protocol will allow every leaving node to become inactive, is the most elaborate. It consists of a series of lemmas. Thus, we first describe the idea of the proof on a high level. For this description, and for the analysis of the theorem, we introduce the notion of a *simplified relay graph*:

**Definition 6.81** (Simplified Relay Graph)**.** *A simplified relay graph $G = (V, S(E))$ of a relay graph $G = (V, E)$ contains the same set of nodes and the following set of edges:*

1. *for every explicit edge $(u, v) \in E$ there is an explicit edge in $S(E)$,*
2. *for every implicit edge $(w, v) \in E$ that is due to a reference of a relay owned by a node $v$ contained in a message in the buffer of some relay $r$, there is an implicit edge $(u, v) \in S(E)$, where $u$ is the sink node of $r$.*

Simply put, in a simplified relay graph the intermediate hops of a message are ignored and each implicit edge is outgoing from the node that will receive the message containing a relay parameter. In legal computation suffixes, these nodes will eventually receive the reference anyway, which is why this definition is helpful.

We now continue with the description of the proof idea: For every computation $C$ we find a sequence of states such that ever more desired properties (which are specified in the following) are fulfilled. We define a leaving node $u$ to be an *anchored node* if *a-out* is a direct relay whose sink node is staying. The sink node of $u$.*a-out* is $u$'s anchor. Note that initially there may not be any anchored node at all. Furthermore, we say a leaving node $u$ is a *towed node* if $u$ is not anchored but $u$ has a staying neighbor in the simplified relay graph. Moreover, we define a leaving

node $u$ to be a *semi-towed node* if $u$ is neither anchored nor towed but $u$ has an anchored neighbor. Note that by the assumption that there is at least one staying node and by Theorem 6.78, there is always at least one towed or semi-towed node as long as there is a non-anchored leaving node. The proof strategy is to show that every towed node will eventually become an anchored node and that every semi-towed node will eventually become a towed node. Inductively, this yields that all active leaving nodes will be anchored at some point in time. From then on, we show that leaving nodes will get rid of both their sink and their non-sink relays (except for the ones stored in *in* and in *a-out*) and all incoming connections until they finally only have a non-sink relay to their anchor, in which case they will execute **stop**. A more specific explanation of the proof structure is given by the lemmas below.

In the following, we will consider an arbitrary but fixed computation and define a series of states according to this computation. Let $S_0$ be an arbitrary state in which the relay layer is in a legal state. Recall that according to Lemma Theorem 6.22 this will be the case throughout $SUFFIX(S_0)$. Furthermore, let $S_1 \geq S_0$ be a state such that all messages in relay buffers during $S_0$ have been received by the sink node and each node has executed the TIMEOUT action at least once after $S_0$. Throughout the rest of this section, we will say a message $m$ in the buffer of some relay $r$ is *received* as soon as it is delivered to the sink node of $r$.

The following is easy to show:

**Lemma 6.82.** *In $SUFFIX(S_1)$, the following invariants hold:*

1. *for every node $u$, every relay owned by $u$ is alive and stored in either $u.N$, $u.D$, $u.in$, $u.a\text{-}out$, or $u.a\text{-}in$,*
2. *for every node $u$, $u.in$ is a sink relay,*
3. *for every node $u$ and every $r \in u.N$, $r$ is direct,*
4. *for every staying node $u$, $u.a\text{-}in = \bot$, $u.a\text{-}out = \bot$,*
5. *for every leaving node $u$, $u.a\text{-}in \neq \bot$, and*
6. *for every leaving node $u$,* has-incoming$(u.a\text{-}out) = false$, direct$(u.a\text{-}out)$, *$u.N = \emptyset$, $u.a\text{-}in$ is a sink relay and if $u.a\text{-}out \neq \bot$ then $u.a\text{-}out$ is a non-sink relay.*

The proof directly follows from the pseudocode at the beginning of the TIMEOUT action (Lines 3–16 and Lines 44–60), the direct relay assumption, the fact that none of the other lines ever violates any of these invariants, and the fact that the protocol deletes a relay only if has-incoming yields false on this relay (which is implicitly the case when the relay was received in a message). Check that all lines marked as dealing with initial states are not executed in $SUFFIX(S_1)$. Thus, in the following we will ignore these lines. For convenience, for a relay $r$ we define $sn(r)$ as the sink node of $r$.

**Lemma 6.83.** *In $SUFFIX(S_1)$, whenever a node $u$ sets $u.a\text{-}out$ to a new value different from $\perp$, $sn(u.a\text{-}out)$ is staying.*

*Proof.* Note that *a-out* is only set to a new value other than $\perp$ in Line 104 and in Line 110 upon receipt of a REVERSE(*out*) message with direct(*out*) = *true* (recall that in the second case, direct(*a-out*) = *true* according to Lemma 6.82 and the semantics of same-target) and only such that afterwards $sn(u.a\text{-}out) = sn(out)$. The only occasions at which REVERSE(*out*) is sent such that *out* is a direct relay in the receiving node are in Line 30 and Line 34, which are only executed by staying nodes (note that although a REVERSE() message is also sent in Line 89, since *a-out* is not a sink relay according to Lemma 6.82 the resulting relay is an indirect relay). Thus, by the definition of $S_1$, for every REVERSE(*out*) message in $SUFFIX(S_1)$ such that direct(*out*) = *true*, $sn(out)$ is staying and the lemma follows. □

**Lemma 6.84.** *There is a state $S_2 \geq S_1$ such that in $SUFFIX(S_2)$ no node $u$ has a relay $r \in u.N$ such that $sn(r)$ is leaving.*

*Proof.* First, assume that there is a node $u$ that adds a relay $r_v$ such that $sn(r_v)$ is leaving to $u.N$ in $SUFFIX(S_1)$. By the direct relay assumption and the pseudocode (note that none of the additional actions we provide adds a relay to the set $N$), this requires that a message of the original protocol containing $r_v$ and sent by $sn(r_v)$ during the execution of an action of the original protocol was received by $u$. However, the REPLACE_A() function would have prohibited the execution of such an action if $sn(r_v)$ is leaving (see Lines 73–77). Thus, in $SUFFIX(S_1)$, no node $u$ adds a relay $r$ to $u.N$ such that $sn(r)$ is leaving.

Second, we show that every relay $r \in u.N$ for some node $u$ such that $sn(r)$ is leaving will be removed from $u.N$ in finite time. Consider a node $u$ such that $r \in u.N$ for some relay $r$ whose sink node $v$ is leaving. According to Lemma 6.82, $u$ must be staying (leaving nodes do not store anything in $N$) and $r$ is direct. By the self-introduction assumption, $u$ will send a message containing a reference to $u.in$ to $v$. Upon receipt, $v$ will send an ASK-TO-REVERSE($r_v$) message such that $r_v$ is the endpoint of $r$ to $u$ (Line 76). Due to Lines 27–29, $u$ will remove all relays to $r_v$ from $N$ (including $r$).

Both parts of this proof together yield that there will be a state $S_2 \geq S_1$ such that no node $u$ stores a relay to a leaving node in $u.N$ and no node will ever do so in $SUFFIX(S_2)$. □

**Lemma 6.85.** *If a node $u$ stores a non-sink relay $r$ in $u.D$ in some state $S$, then there will be a state $S' \geq S$ such that $r$ has been deleted.*

*Proof.* Assume there is a non-sink relay $r$ stored by node $u$ in $u.D$ in some state $S$. Let $Source(r)$ be the set that contains all relays $r'$ such that has-incoming($r'$) = $false$ and such that there is a directed path via relays to $r$ in the relay graph (intuitively, $Source(r)$ contains the leaves of the relay subtree rooted at $r$). To simplify the description, we let $Source(r)$ also contain all relays $r'$ fulfilling the above conditions that will be created upon receipt of a message because of a relay

reference contained in that message. We define the following potential function $\Phi_r$: $\Phi_r := \sum_{r' \in Source(r)}(r'.level = r.level)$. We will show that $\Phi_r$ never increases and, as long as $\Phi_r > 0$, it will always decrease in finite time.

First of all, note that every relay in $Source(r)$ is an indirect relay. Thus, no node $v$ stores a relay from $Source(r)$ in $v.N$ according to Lemma 6.82. Furthermore, for the same reason, no relay from $Source(r)$ is stored in $v.N$ for any node $v$. One can check in the pseudocode that in the actions added by the transformation only the references of direct or sink relays are sent in a message. Since the actions of the original protocol only access the relays in $v.N$, we obtain that $\Phi_r$ never increases.

To show that $\Phi_r$ will decrease in finite time as long as $\Phi_r > 0$, consider an arbitrary node $v$ that owns a relay $r' \in Source(r)$ or that will receive a message upon whose receipt $r' \in Source(r)$ will be created. We now show that $v$ will delete $r'$. If $r'$ is the parameter of a message received by $v$, $r'$ will be deleted upon receipt of this message due to the direct relay assumption and the pseudocode. Otherwise, $r'$ must be stored in $v.D$ because of Lemma 6.82 and the fact that direct$(r') = false$. Then, during TIMEOUT, $v$ will delete $r'$ either in Line 20 or Line 65 (depending on whether $v$ is staying or leaving). Thus, $r'$ will be completely deleted and $\Phi_r$ will decrease in finite time (note that even if an additional relay $r''$ belongs to $Source(r)$ after the deletion because has-incoming$(r'')$ has become false, the level of $r''$ must be smaller than the level of $r'$).

Thus, $\Phi_r$ will eventually be zero, and $u$ will delete $r$ during TIMEOUT in Line 20 or Line 65 (depending on whether $u$ is staying or leaving). □

**Lemma 6.86.** *There is a state $S_3 \geq S_2$ such that in $SUFFIX(S_3)$ for every leaving node $u$ no node will store a relay whose sink is $u.a\text{-}in$ (in any of its variables), and there is no active node $v$ such that $v.a\text{-}out \neq \perp$ and the sink node of $v.a\text{-}out$ is leaving.*

*Proof.* Consider an arbitrary leaving node $u$ and define the potential $\Phi_u$ as the number of variables (over all nodes) that store a relay whose sink is $u.a\text{-}in$. We first show that $\Phi_u$ is monotonically decreasing and then prove that it decreases in finite time as long as $\Phi_u > 0$.

Note that in $SUFFIX(S_2)$ no node $v$ adds a relay whose sink node is leaving to $v.D$: Whenever in the pseudocode a relay $r$ is added to $v.D$ in $SUFFIX(S_2)$ (in which the lines to deal with the initial states are not executed), it has been in $v.N$ before or it was $v.a\text{-}in$ before. In the former case, the sink of $r$ cannot have been leaving according to Lemma 6.84; in the latter case, the relay is a sink relay according to Lemma 6.82. Thus, Lemma 6.85 implies that eventually no node $v$ will store a relay to $u.a\text{-}in$ in $v.D$ for any leaving node $u$. Therefore, eventually the only relay to $u.a\text{-}in$ that is stored in a variable of a node $v$ must be stored in $v.a\text{-}out$ (and $v$ must thus be a leaving node). Note that any node $v$ only adds a relay $r$ to $v.a\text{-}out$ if it receives a REVERSE$(r)$ message such that direct$(r) = true$ (see Line 104). Such a message can have been sent by the owner of $r$ only. However, a REVERSE() message is sent by a leaving node only in Line 89, with a parameter

that is a reference to a non-sink relay according to Lemma 6.82. Thus and since corrupted messages originally in the system have been received in $S_1$ already, $\Phi_u$ is monotonically decreasing.

Recall that we have shown that every node $v$ storing a relay with sink $u.a\text{-}in$ stores this relay in $v.a\text{-}out$. During TIMEOUT, $v$ will send a NOTIFY-ANCHOR() message to $u$ (see Line 71), causing $u$ to renew $u.a\text{-}in$ (Line 99) after which $\Phi_u = 0$ trivially holds. Since $\Phi_u$ is monotonically decreasing, there cannot be any node that stores a variable with a relay whose sink is $u.a\text{-}in$ after that point in time. Since $u$ was chosen arbitrarily, there is a state $S_2' \geq S_2$ such that in $SUFFIX(S_2')$ for every leaving node $u$ no node will store a relay whose sink is $u.a\text{-}in$.

Note that in the above we only showed that there will be no explicit edge with $u.a\text{-}in$ as an endpoint in the relay graph in $SUFFIX(S_2')$. As a next step, we will prove that for every active leaving node $v$ that never gets inactive, has-incoming$(v.a\text{-}in) = false$ in an infinite number of states. This will be used to prove the second part of the lemma. In the following, we say a relay $r$ is *undesired* if and only if $r$'s sink is $u.a\text{-}in$ for any node $u$. A message is *undesired* if and only if it contains a parameter of an undesired relay. Note that due to Lemma 6.84, there is a state $S_2'' \geq S_2'$ such that no leaving node will receive a message of the original protocol: i.e., the action REPLACE_A() will not be executed during $SUFFIX(S_2'')$ by leaving nodes. Furthermore, note that the ASK-TO-REVERSE-ANCHOR() message is sent only via relay that is stored by the sender. Thus, there is a state $S_2''' \geq S_2''$ such that no leaving node $v$ will receive an ASK-TO-REVERSE-ANCHOR() message via $v.a\text{-}in$ in $SUFFIX(S_2''')$. We can combine these two facts to obtain that in $SUFFIX(S_2''')$ no undesired ASK-TO-REVERSE($r$) message is sent (*). Therefore, check in the pseudocode that the only places where an undesired ASK-TO-REVERSE() message is sent is in the REPLACE_A() action of a leaving node (Line 76) and when an ASK-TO-REVERSE-ANCHOR() (Line 94) message was received via $v.a\text{-}in$. From Lemma 6.84 we know that no staying node sends an undesired message in $SUFFIX(S_2''')$. Furthermore, from (*) we know that no undesired ASK-TO-REVERSE() message is sent in $SUFFIX(S_2''')$. It follows from the pseudocode that the only undesired message that could be sent in $SUFFIX(S_2''')$ is an ASK-TO-REVERSE-ANCHOR() message that is sent in Line 70. This line, however, is executed by a node $v$ only if has-incoming$(v.a\text{-}in) = false$. Since this is the only occasion at which an ASK-TO-REVERSE($r$) message such that $r$'s sink is $v.a\text{-}in$ is sent in $SUFFIX(S_2''')$, has-incoming$(v.a\text{-}in) = false$ holds in an infinite number of states (it always holds when the existing ASK-TO-REVERSE($v.a\text{-}in$) message has been received until $v$'s next execution of TIMEOUT).

To finish the proof of the second part of the lemma's claim, consider an arbitrary but fixed active node $v$ such that $v.a\text{-}out$ is a direct non-sink relay to some relay $r'$ owned by a leaving node. Note that $v$ must be leaving according to Lemma 6.82. As we have shown before, at some state has-incoming$(v.a\text{-}in) = false$ will hold. We also proved that $v$ does not send a reference of $v.a\text{-}in$ except for in the TIMEOUT action. Thus, upon the next execution of TIMEOUT, has-incoming$(v.a\text{-}in) = false$ still holds and $v$ sends an ASK-TO-REVERSE-ANCHOR($v.a\text{-}in$) message to

$sn(v.a\text{-}out)$ due to Line 70. Upon receipt of this message, $sn(v.a\text{-}out)$ will send ASK-TO-REVERSE($r$) to $v$ such that $r$ is the sink of $v.a\text{-}out$ due to Line 94. Upon receipt, this message will cause $v$ to set $v.a\text{-}out$ to $\perp$ (Line 86). This yields a contradiction. Together with Lemma 6.83, we obtain that there is a state $S_3$ as specified in the lemma. □

**Lemma 6.87.** *There is a state $S_4 \geq S_3$ such that in $SUFFIX(S_4)$ no node will change the value of a-out to $\perp$ or change the value of a-in.*

*Proof.* Define $S_4 \geq S_3$ as the state in which all ASK-TO-REVERSE() and all NOTIFY-ANCHOR() messages in buffers in $S_3$ have been received.

For the first claim of the lemma, consider an arbitrary node $u$ such $a.a\text{-}out \neq \perp$. Observe in the pseudocode that in $SUFFIX(S_1)$ the only occasion at which $u$ could possibly set $u.a\text{-}out$ to $\perp$ is in Line 86 upon receipt of an ASK-TO-REVERSE($out$) message such that same-target$(out, u.a\text{-}out) = true$. We now show that such a message cannot exist in $SUFFIX(S_4)$. Observe in the pseudocode that staying nodes never send any ASK-TO-REVERSE() message. Since every relay reference contained in a message must be a reference to a relay owned by the sender, in $SUFFIX(S_4)$ there thus cannot be any ASK-TO-REVERSE($out$) message in any buffer such that $out$ is a reference to a relay owned by a staying node. Lemma 6.86 implies that the relay that $u.a\text{-}out$ is connected to is owned by a staying node $v$ in $SUFFIX(S_3)$. Thus, same-target$(out, u.a\text{-}out)$ must always be false when ASK-TO-REVERSE($out$) is executed during $SUFFIX(S_4)$ and the claim is proven.

For the second claim, consider an arbitrary node $u$ such that $u.a\text{-}in \neq \perp$. Observe in the pseudocode that in $SUFFIX(S_1)$ the only occasion at which $u$ could possibly change the value of $u.a\text{-}in$ is in Line 99. This line is executed only if $u$ received a NOTIFY-ANCHOR() message. We now show that such a message cannot exist in $SUFFIX(S_4)$. Due to the definition of $S_4$, every NOTIFY-ANCHOR() message existing in any state of $SUFFIX(S_4)$ must have been sent in $SUFFIX(S_3)$. Note that according to the pseudocode such a message is only sent in Line 71 and only to $v.a\text{-}out$, where $v$ is the sending node. According to Lemma 6.86, $v.a\text{-}out$ is owned by a staying node in $SUFFIX(S_3)$, thus no such message can be sent to $u$ in $SUFFIX(S_3)$. Therefore, no such message exists in $SUFFIX(S_4)$ and the claim is proven. □

Note that the fact that a leaving node $u$ never adds a relay to $u.D$ except for in Line 98, where it subsequently changes the value of $u.a\text{-}in$, and Lemma 6.87 imply:

**Corollary 6.88.** *There is a state $S_x \geq S_4$ such that in $SUFFIX(S_x)$, for every active leaving node $u$, $u.D = \emptyset$.*

**Lemma 6.89.** *There is a state $S_5 \geq S_x$ such that in $SUFFIX(S_5)$ there will be no* REVERSE() *message or message of the original protocol that contains a reference of a relay whose sink node is leaving. Additionally, in $SUFFIX(S_5)$, no node stores a relay $r$ such that the sink node of $r$ is leaving.*

*Proof.* Define $S'_x \geq S_x$ as the state in which all REVERSE() messages and all messages of the original protocol that were in a relay buffer in $S_4$ have been received. We first show that in $SUFFIX(S'_x)$ there will be no REVERSE() message and no message of the original protocol containing a reference to a relay $r$ owned by a leaving node.

First, assume for contradiction that in $SUFFIX(S_x)$ a REVERSE($r$) message is delivered to a leaving node $u$. However, in the pseudocode any REVERSE() message is sent either with parameter *a-out*, whose sink node is staying due to Lemma 6.86, or with parameter *in* and by a staying node. Second, note that no leaving node sends a message of the original protocol (since leaving nodes do not execute the original protocol actions). Whenever a staying node $u$, however, sends a message of the original protocol, any reference to a relay contained in the parameter list of that message must be a reference to a relay in $u.N \cup \{u.in\}$. According to Lemma 6.84, the sink nodes of all such references are staying then.

The fact that no node stores a relay to a leaving node in *a-out* follows from Lemma 6.86 and the fact that no node $u$ stores a relay to a leaving node in $u.N$ follows from Lemma 6.84. Due to Lemma 6.85, all that remains to be shown is that no node $u$ will ever add a relay $r_v$ to a leaving node to $u.D$. To this end, we go through all different cases where a relay is added to $D$. Lines 3–16 and Lines 44–60, are never executed in $SUFFIX(S_1)$ according to Lemma 6.82. In Line 29, a relay $r$ is only added to $D$ if it was previously stored in $N$, which cannot be the case for a relay such that $sn(r)$ is leaving according to Lemma 6.84. Last, Line 98 is executed only if $u$ is leaving and upon receipt of a NOTIFY-ANCHOR() message, which is only sent by a node $w$ in Line 71 to $sn(w.a\text{-}out)$. If such a NOTIFY-ANCHOR() message is received in $SUFFIX(S'_x)$, it must have been sent in $SUFFIX(S_x)$. According to Lemma 6.86, the sink node of $w.a\text{-}out$ must have been staying at that time, which implies that the message cannot have been sent via a relay whose sink node is $u$. Thus, eventually for every relay $u$, $u.D$ will not contain any relay whose sink node is leaving. All in all, since for every node $u$ there are no other relays with outgoing connection than *a-out*, those in $u.N$ and those in $u.D$ after $S_1$, the proof of the claim is finished. □

**Lemma 6.90.** *For each leaving node $u$ that is towed in a state $S \geq S_5$, there is state $S' \geq S$ such that $u$ is anchored in $S'$.*

*Proof.* Consider an arbitrary node $u$ that is towed in some state $S \geq S_5$. We show that $u$ will become anchored in $SUFFIX(S_5)$. Since $u$ is towed in state $S$, $u$ must have a staying neighbor $v$ in the simplified relay graph in $S$. This means that there is a (possibly implicit) edge $(v, u)$ or an edge $(u, v)$ in the simplified relay graph in $S$. We consider the two cases individually.

For the first case, assume that $v$ has an edge in the simplified relay graph to $u$. According to Lemma 6.89 this edge must be implicit and it must be due to an ASK-TO-REVERSE($r$) message or an ASK-TO-REVERSE-ANCHOR($r$) message such that $r$ is owned by $u$. In both cases, since $v$ is staying, it will send a REVERSE($v.in$)

message to $r$ (check Lines 30 and 34). Upon receipt, $u$ will update *a-out* to the received message and thus become anchored.

For the second case, assume that $u$ has an edge in the simplified relay graph to $v$. According to Lemma 6.82 and Corollary 6.88; the fact that *a-out* is empty (otherwise, $u$ would be anchored according to Lemma 6.86), this edge must be implicit. If the implicit edge is due to a message of the original protocol or due to an ASK-TO-REVERSE-ANCHOR() message, $u$ will send an ASK-TO-REVERSE($r$) message such that $r$ is owned by $u$ to $v$ in Line 76 or Line 94. If the implicit edge is due to an ASK-TO-REVERSE() message, $u$ will send an ASK-TO-REVERSE($u.in$) message to $v$ in Line 81. Last, if the implicit edge is due to a REVERSE(*out*) message then if direct(*out*) = *true*, $u$ will set *a-out* to *out* in Line 104 or send an ASK-TO-REVERSE($u.in$) message to $v$ in Line 106. Thus, in any case, either $u$ becomes anchored immediately, or we end up in the first case for which we have already proven that it leads to $u$ becoming anchored. Thus, the proof of the lemma is finished. □

**Lemma 6.91.** *For each leaving node $u$ that is semi-towed in a state $S \geq S_5$, there is a state $S' \geq S$ such that $u$ is towed in $S'$.*

*Proof.* Consider an arbitrary node $u$ that is semi-towed in some state $S \geq S_5$. We show that $u$ will become towed in $SUFFIX(S_5)$. Since $u$ is semi-towed in state $S$, $u$ must have an anchored leaving neighbor $v$ in the simplified relay graph in $S$. This means that there is a (possibly implicit) edge $(v, u)$ or an edge $(u, v)$ in the simplified relay graph in $S$. We consider the two cases individually.

First of all, assume that $v$ has an edge in the simplified relay graph to $u$. According to Lemma 6.89 this edge must be implicit and it must be due to an ASK-TO-REVERSE($r$) message (*) or an ASK-TO-REVERSE-ANCHOR($r$) message such that $r$ is owned by $u$. In the former case, upon receipt of the message $v$ sends a REVERSE($v.a\text{-}out$) message to $u$ (see Line 89) because same-target($v.a\text{-}out, r$) must be false according to Lemma 6.86. This causes $u$ to be towed afterwards. In the latter case, $v$ sends an ASK-TO-REVERSE($r'$) message to $u$ such that $r'$ is owned by $v$ (see Line 94). Upon receipt of this message, $u$ sends an ASK-TO-REVERSE($u.in$) message to $v$ and we end up in a case we already considered (see (*)).

Second, assume that $u$ has an edge in the simplified relay graph to $v$. Again, according to Lemma 6.89 this edge must be implicit and it must be due to an ASK-TO-REVERSE($r$) message or an ASK-TO-REVERSE-ANCHOR($r$) message such that $r$ is owned by $v$. In both cases, $u$ will respond with an ASK-TO-REVERSE($r$) message such that $r$ is owned by $u$ (see Lines 81 and 94) and we are in a case we already considered again (see (*)). Thus we obtain the claim of the lemma. □

Lemma 6.90, Lemma 6.91, the fact that there cannot remain a leaving node that is neither anchored nor towed due to Theorem 6.78 and the assumption that at least one node is staying yield the following corollary:

**Corollary 6.92.** *There is a state $S_6 \geq S_5$ such that every active leaving node is anchored in every state of $SUFFIX(S_6)$.*

**Lemma 6.93.** *There is a state $S_8 \geq S_7$ such that in $SUFFIX(S_8)$ for every leaving node $u$ there is at most one incoming connection to $u.a\text{-}in$, which is due to an* Ask-to-reverse-anchor($u.a\text{-}in$) *message in $u.a\text{-}out.Buf$, and there is no incoming connection to $u.in$.*

*Proof.* Consider an arbitrary but fixed leaving node $u$ with has-incoming($u.in$) = *true* in $S_7$. Note that according to Lemma 6.89 any incoming connection to a relay owned by $u$ may only be due to an Ask-to-reverse($r_u$) message with $sn(r_u) = u$ or an Ask-to-reverse-anchor($r_u$) message with $sn(r_u) = u$.

First, note that in $SUFFIX(S_7)$, no Ask-to-reverse($r_u$) message such that $sn(r_u) = u$ is created (Line 63 is not executed due to Corollary 6.88, Lines 81 and 106 are not executed due to Corollary 6.92, and Line 112 creates an Ask-to-reverse() message whose parameter is a relay to a staying node by Lemma 6.86). Furthermore, whenever an Ask-to-reverse($r_u$) message such that $sn(r_u) = u$ is received, $r_u$ is not sent in another message but deleted in either Line 31, Line 82 or Line 90 (note that Line 85 cannot be executed because same-target($r_u, a\text{-}out$) must be false since $a\text{-}out$ must be direct according to Lemma 6.82 and owned by a staying node according to Lemma 6.86). Thus, there is a state $S_7'$ such that in $SUFFIX(S_7')$, there will be no Ask-to-reverse($r_u$) message with $sn(r_u) = u$.

Second, note that in $SUFFIX(S_7)$, an Ask-to-reverse-anchor($r_u$) message with $sn(r_u) = u$ is created only if has-incoming($u.a\text{-}in$) = *false* and only with parameter $u.a\text{-}in$ and only sent to the sink node of $u.a\text{-}out$ (see Lines 68–70 and Lemma 6.86). Furthermore, for every Ask-to-reverse-anchor($r_u$) message with $sn(r_u) = u$ in any buffer, when the corresponding action is executed, $r_u$ is not sent in another message but deleted in Line 95. Thus, there is a state $S_7'' \geq S_7$ such that in $SUFFIX(S_7'')$ there will always be at most one Ask-to-reverse-anchor($r_u$) message with $sn(r_u) = u$, which is in the message channel of a staying node and the endpoint of $r_u$ must be $u.a\text{-}in$.

Let $S_8$ be the latter of the two states $S_7'$ and $S_7''$. Then $S_8$ fulfills the requirements of the lemma. □

**Lemma 6.94.** *There is a state $S_9 \geq S_8$ such that every leaving node is inactive in $S_9$.*

*Proof.* Consider an arbitrary leaving node $u$ in $SUFFIX(S_8)$. Note that **stop** is executed in Line 67 in Timeout as soon as the conditions of Line 66 are fulfilled. Thus, according to Corollary 6.88 and Lemma 6.93, the only reason for **stop** not to be executed by $u$ is that has-incoming($u.a\text{-}in$) = *true* all the time, which must be due to an Ask-to-reverse-anchor($r_u$) message such that $sn(r_u) = u$ in the message channel of the node $v$ with $v = sn(u.a\text{-}out)$, which is staying according to Lemma 6.86. Upon receipt of that message, that node will send a Reverse($v.in$) message to $u.a\text{-}in$ and delete $r_u$ (see Line 34). Upon receipt, the Reverse($v.in$)

message will cause $u$ to merge the relay created due to the message receipt with $u.a\text{-}out$ (Line 110). Thus, $u$ does not have any incoming connection afterwards and the only outgoing connection is stored in $u.a\text{-}out$. Upon the next execution of TIMEOUT, the conditions of Line 66 are fulfilled and $u$ will execute **stop**.

Since $u$ was chosen arbitrarily, every leaving node will eventually execute **stop** and there is a state $S_9$ as specified in the lemma. □

From Lemma 6.94 and the fact that no staying node ever executes **stop**, we immediately obtain Theorem 6.79, restated as follows:

**Theorem 6.79.** *In every computation of the transformed protocol in which there is at least one staying node, every leaving node will eventually be inactive and every staying node will always remain active.*

### Proof of Theorem 6.80 - Eventual Original Protocol Behavior

Building on the analysis of Theorem 6.79, one can easily show that in every computation there is a state after which the transformed protocol behaves exactly as the original one: i.e., the overall stabilization to a certain topology is not harmed by the transformation. This is formalized by Theorem 6.80, which we restate as follows:

**Theorem 6.80.** *In every computation $C$ of the transformed protocol, there is a state $S$ such that every leaving node is inactive, every staying node is active and there is a computation $C'$ of the original protocol starting from $S$ such that $C' = SUFFIX_C(S)$.*

*Proof.* First of all, note that the only additional variable of a staying node introduced by the transformation of the protocol is $D$, which will be empty for every node at some state $S_{10} \geq S_9$ according to Lemma 6.85 and the direct relay assumption (note that since no node permanently stores an indirect relay, it will never be moved to $D$ during TIMEOUT). Furthermore, REPLACE_A() behaves exactly as $A$ after $S_9$. Among the other actions introduced by the transformation, none of them is called during TIMEOUT after $S_9$, because no node is leaving anymore. For staying nodes, each of the other actions introduced by the transformation does not cause any other than a REVERSE() message and this message does not cause any other message. Thus, after all these messages still in the system have been received, a state as desired in the Theorem is reached. □

PART

# IV

# Conclusion

CHAPTER 7

# Applications and Open Research Questions

In each of the main chapters of this thesis, local graph transformation primitives played an important role. In Chapter 3, we investigated the complexity of transforming graphs with a minimum number of applications of primitives from $\mathcal{IDF}$ or $\mathcal{IDFR}$. In Chapter 4, an adapted set of graph transformation primitives, $\mathcal{ISF}$ turned out to be helpful to enable monotonic searchability. In Chapter 5, although not mentioned explicitly, the given protocol manipulated edges only according to the primitives of $\mathcal{IDF}$ to obtain a solution to the $\mathcal{FDP}$ in conjunction with monotonic searchability. Last, in Chapter 6 the set $\mathcal{IFR}$ of graph transformation primitives for the relay model was shown to emulate each of the primitives in $\mathcal{IDFR}$, which helped to prove the universality of the relay model and to transform existing protocols to the relay model.

In this chapter, we discuss further possible applications of the results of this thesis and summarize ideas for future research.

### NP-Hardness and Approximability of Local Graph Transformations

Due to the importance of local graph transformations, which this thesis has highlighted, the problem of transforming graphs by a minimum number of primitive applications is an interesting theoretical question on its own. However, in the context of so-called supervised overlay networks, there is also an interesting immediate application of the problem posed in Chapter 3. In a *supervised overlay network* there is a dedicated, trusted node called *supervisor*. This node controls all network adaptations but otherwise is not involved in the functionality of the overlay network (such as serving search requests), which is handled in a peer-to-peer manner. This has the advantage that even if the supervisor is down, the overlay network is still functional. Of course, a malicious supervisor would pose a significant problem for an overlay network, since it could easily launch *Sybil attacks* (i.e., flood the overlay network with fake or adversarial nodes) or *Eclipse attacks* (i.e., isolate nodes from other nodes in the overlay network). It is thus crucial to limit the power of a supervisor in order to prevent such attacks. A possible approach to do this is to restrict the supervisor such that every command issued to the nodes must resemble an application of a primitive in $\mathcal{IDF}$ or $\mathcal{IDFR}$. This way, the supervisor can still transform every weakly connected topology into any other (weakly) connected topology (since the primitives are universal), but cannot disconnect the network (since the primitives preserve weak connectivity). With a slight additional assumption, the supervisor also cannot introduce new nodes into the network: The assumption we make is that all connections are *authorized*,

meaning that both endpoints are aware of the other endpoint of this connection. If, for an edge $(u, v)$ that is supposed to be transformed into $(v, u)$ by an application of the reversal primitive, $v$ verifies that $u$ actually was the previous endpoint of the former edge, then the primitives cannot be used to introduce new nodes into the network. In such a setting it is certainly desirable to transform topologies with a minimum number of primitive applications, since this reduces the communication work of the supervisor as well as the overall communication among the nodes.

Of course, this description makes some assumptions that may be too strong in practical scenarios. Most notably, we assumed that only the server could act maliciously but that the participants of the network are honest and correct: i.e., they refuse any graph transformation commands beyond the four primitives. What, however, if some participants also behave in a malicious manner? Is it still possible to avoid Sybil or Eclipse attacks? This is an open question that might be considered in future research. At first glance, it seems that in this case the only measure that would help is to form quorums of nodes that are sufficiently large so that at least one node in each quorum is honest.

Even from a purely theoretical view, our results give rise to additional questions: For example, does the NP-hardness apply to any set of local graph transformation primitives, or is there a set of local graph transformation primitives that can transform arbitrary initial graphs much faster into arbitrary final graphs than the set considered in this thesis? Furthermore, is it possible to obtain decentralized versions of the approximation algorithms presented in Chapter 3 and, if so, what is their competitiveness when compared to the centralized ones? Moreover, given that for practical purposes it might be reasonable to assume that the desired topology is subject to change, is it possible to obtain good online algorithms for the local graph transformation problem? Besides these considerations, it would be interesting to study variants of the problem. Given the supervised overlay network setting, for example, it is reasonable to assume that independent primitive applications could be performed in parallel. Of course, one could simply apply as many primitives as possible from a sequence of applications computed by one of the approximation algorithms in Chapter 3 in each step to speed up the overall transformation. Yet, it would be worth investigating whether shorter sequences are polynomially computable for this problem variant.

### Monotonic Searchability for Supergraphs of the Line and Monotonic Searchability under Leaving Nodes

Being able to search reliably during the stabilization phase in self-stabilizing systems is an important feature, especially because it is generally impossible to locally determine the point in time when the system has stabilized. In fact, what we called search in this work generally describes sending messages to another node to which a direct connection does not exist. Hence, monotonic searchability naturally has many applications in distributed systems and overlay networks and is one of the most basic problems in this field.

The same applies to the finite departure problem. In large distributed systems, a high turnover is not the exception but the norm. Whereas the arrival of a new node does not need to be considered explicitly for self-stabilizing topologies (since according to the definition of self-stabilization, the node will be integrated by the self-stabilizing protocol), taking account of node departures is much more involved. This has to do with the fact that the overall system's behavior must not be compromised due to one or more nodes leaving.

Although our results of Chapter 4 solve the problem of monotonic searchability for a wide range of topologies, there are certain aspects that have not been studied yet. For example, we did not consider the additional cost of convergence (i.e., the amount of additional messages to be sent), nor the impact of our methods on the convergence time of the topology. Additionally, while our generic search protocol enables us to search existing nodes in legal states with a low dilation, searching for a non-existing node can still cause a message to travel $\Omega(n)$ hops, even in legal states. Whether this is provably necessary or could be improved is still an open question.

As mentioned before, we deliberately restricted our considerations regarding monotonic searchability under node departures in Chapter 5 to the line topology, since this turned out to be very complex already. Anyway, it would be interesting to investigate whether a general approach for the conjunction of the two problems can also be obtained and what this would look like.

### The Relay Model and Its Self-Stabilizing Realization

Although the original goal of the development of the relay model introduced in Chapter 6 was to have a reasonable model that permits a solution to the $\mathcal{FDP}$ without oracles, it has numerous additional advantages that make it useful in a large number of applications.

For instance, observe that the relay model offers facilities for admission control that the traditional interconnection model does not offer: In the standard model, possessing a reference to another node $u$ admits a node to send a message to $u$, and $u$ is unable to revoke this right. In the relay model, in contrast, each node is able to delete a relay and thus revoke the right to send a message to this relay. Even worse in the standard model, a reference can be copied and introduced to other nodes without the permission of $u$. Although it is also possible to forward a relay reference in the relay model, such an action only establishes an indirect connection. To create a direct relay connection, a permission from $u$ would still be required. This has a huge advantage in the scenario of distributed denial-of-service attacks if we assume that the attacker has no access to the relay layer (which may be reasonable if the relay layer is implemented using secure hardware): If an attacker $v$ forwards a relay reference $\hat{r}$ to other nodes in order to attack the sink node of $r$, the bandwidth of the attack is not the sum of the individual bandwidths of all participating nodes, but limited by the bandwidth of $v$. Thus, forwarding the relays does not yield any advantage for the attack.

The relay model may also be useful in the area of anonymous communication. This is due to the fact that nodes can create multiple relays as pseudonyms. Whereas in the original model each node was uniquely defined by its reference that was even propagated to the application layer, in the relay model applications only know locally valid references. Although applications can check whether two relays have the same next target, it is not possible for them to determine whether the sink node of two relays with different next targets is equal. Therefore, a node could use different sink relays for different purposes in the network and no other nodes would be able to link this node's activities. This way, anonymity would be achieved.

An important open research question is how to realize the relay layer in practice. One possibility is to use so-called middleboxes that are placed between the computer and the network and run the relay layer. Due to being a dedicated piece of hardware, these middleboxes could be secured effectively and the implementation of the relay layer could be as minimalistic and verified as possible. An additional benefit of this solution is that the relay layer of a computer could continue to run even if the computer crashes or experiences a power failure (for the latter the middlebox could be equipped with a battery). Despite being very secure and failsafe, this approach has the downside of being very costly since it requires every computer in the network to be equipped with a middlebox running the relay layer. For this reason, it might also be worthwhile investigating possibilities to implement the relay layer using existing hardware at the price of giving weaker security guarantees. One possible such approach is to rely on secure enclaves offered by modern central processing units. An example of this is Intel's Software Guard Extensions (SGX). Unfortunately, this technique cannot prevent an attacker who gained control over the operating system from dropping packages received via the network card. It is thus an interesting open research question whether or to which extent cryptographic methods can help to limit the power of an attacker in this approach.

# Bibliography


[AAB04] Baruch Awerbuch, Yossi Azar, and Yair Bartal. **On-line generalized Steiner problem**. In: *Theoretical Computer Science* 324.2-3 (2004), pp. 313–324. DOI: 10.1016/j.tcs.2004.05.021.

[AKR95] Ajit Agrawal, Philip N. Klein, and R. Ravi. **When Trees Collide: An Approximation Algorithm for the Generalized Steiner Problem on Networks**. In: *SIAM Journal on Computing* 24.3 (1995), pp. 440–456. DOI: 10.1137/S0097539792236237.

[And+01] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. **Resilient Overlay Networks**. In: *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*. Banff, Alberta, Canada, 2001, pp. 131–145. DOI: 10.1145/502034.502048.

[And+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. **Graph Transformation for Specification and Programming**. In: *Science of Computer Programming* 34.1 (1999), pp. 1–54. DOI: 10.1016/S0167-6423(98)00023-9.

[AS07] James Aspnes and Gauri Shah. **Skip graphs**. In: *ACM Transactions on Algorithms* 3.4 (2007), pp. 37:1–37:25. DOI: 10.1145/1290672.1290674.

[ASK18] Saadia Albane, Hachem Slimani, and Hamamache Kheddouci. **Graph grammars according to the type of input and manipulated data: A survey**. In: *Computer Science Review* 28 (2018), pp. 178–203. DOI: 10.1016/j.cosrev.2018.04.001.

[AW07] James Aspnes and Yinghua Wu. **O(logn)-Time Overlay Network Construction from Graphs with Out-Degree 1**. In: *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS)*. Guadeloupe, French West Indies, 2007, pp. 286–300. DOI: 10.1007/978-3-540-77096-1_21.

[BC97] Piotr Berman and Chris Coulston. **On-Line Algorithms for Steiner Tree Problems (Extended Abstract)**. In: *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC)*. El Paso, Texas, USA, 1997, pp. 344–353. DOI: 10.1145/258533.258618.

[Ben+18] Markus Benter, Till Knollmann, Friedhelm Meyer auf der Heide, Alexander Setzer, and Jannik Sundermeier. **A Peer-to-Peer Based Cloud Storage Supporting Orthogonal Range Queries of Arbitrary Dimension**. In: *Revised Selected Papers of the 4th International Symposium on Algorithmic Aspects of Cloud Computing (ALGOCLOUD)*. Helsinki, Finland, 2018, pp. 46–58. DOI: 10.1007/978-3-030-19759-9_4.

[BGP13] Andrew Berns, Sukumar Ghosh, and Sriram V. Pemmaraju. **Building self-stabilizing overlay networks with the transitive closure framework**. In: *Theoretical Computer Science* 512 (2013), pp. 2–14. DOI: 10.1016/j.tcs.2013.02.021.

[Bui+07] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. **Snap-stabilization and PIF in tree networks**. In: *Distributed Computing* 20.1 (2007), pp. 3–19. DOI: 10.1007/s00446-007-0030-4.

[Coo71] Stephen A. Cook. **The Complexity of Theorem-proving Procedures**. In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC)*. Shaker Heights, Ohio, USA, 1971, pp. 151–158. DOI: 10.1145/800157.805047.

[Cor+12] Andreas Cord-Landwehr, Martina Hüllmann, Peter Kling, and Alexander Setzer. **Basic Network Creation Games with Communication Interests**. In: *Proceedings of the 5th International Symposium on Algorithmic Game Theory (SAGT)*. Barcelona, Spain, 2012, pp. 72–83. DOI: 10.1007/978-3-642-33996-7_7.

[CT96] Tushar Deepak Chandra and Sam Toueg. **Unreliable Failure Detectors for Reliable Distributed Systems**. In: *Journal of the ACM* 43.2 (1996), pp. 225–267. DOI: 10.1145/226643.226647.

[Del+07] Nelly Delessy, Eduardo B. Fernandez, Maria M. Larrondo-Petrie, and Jie Wu. **Patterns for Access Control in Distributed Systems**. In: *Proceedings of the 14th Conference on Pattern Languages of Programs*. Monticello, Illinois, USA, 2007, pp. 3:1–3:11. DOI: 10.1145/1772070.1772074.

[Del+10] Sylvie Delaët, Stéphane Devismes, Mikhail Nesterenko, and Sébastien Tixeuil. **Snap-stabilization in message-passing systems**. In: *Journal of Parallel and Distributed Computing* 70.12 (2010), pp. 1220–1230. DOI: 10.1016/j.jpdc.2010.04.002.

[DH97] Shlomi Dolev and Ted Herman. **Superstabilizing Protocols for Dynamic Distributed Systems**. In: *Chicago Journal of Theoretical Computer Science* 1997 (1997). DOI: 10.4086/cjtcs.1997.004.

[Dij74] Edsger W. Dijkstra. **Self-stabilizing Systems in Spite of Distributed Control**. In: *Communications of the ACM* 17.11 (1974), pp. 643–644. DOI: 10.1145/361179.361202.

[DK08] Shlomi Dolev and Ronen I. Kat. **HyperTree for self-stabilizing peer-to-peer systems**. In: *Distributed Computing* 20.5 (2008), pp. 375–388. DOI: 10.1007/s00446-007-0038-9.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. **Tor: The Second-Generation Onion Router**. In: *Proceedings of the 13th USENIX Security Symposium (USENIX)*. San Diego, California, USA, 2004, pp. 303–320.

[Dol+11] Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. **Stabilizing data-link over non-FIFO channels with optimal fault-resilience**. In: *Information Processing Letters* 111.18 (2011), pp. 912–920. DOI: 10.1016/j.ipl.2011.06.010.

[Dol+12] Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. **Self-stabilizing End-to-End Communication in (Bounded Capacity, Omitting, Duplicating and non-FIFO) Dynamic Networks**. In: *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Toronto, Ontario, Canada, 2012, pp. 133–147. DOI: 10.1007/978-3-642-33536-5_14.

[Dol00] Shlomi Dolev. **Self-Stabilization**. MIT Press, 2000. DOI: 10.7551/mitpress/6156.001.0001.

[DT13] Shlomi Dolev and Nir Tzachar. **Spanders: Distributed spanning expanders**. In: *Science of Computer Programming* 78.5 (2013), pp. 544–555. DOI: 10.1016/j.scico.2012.10.001.

[Ehr+99] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, eds. **Handbook of Graph Grammars and Computing by Graph Transformations, Volume 3: Concurrency, Parallelism, and Distribution**. World Scientific, 1999. DOI: 10.1142/4181.

[ESS14a] Martina Eikel, Christian Scheideler, and Alexander Setzer. **Minimum Linear Arrangement of Series-Parallel Graphs**. In: *Revised Selected Papers of the 12th International Workshop on Approximation and Online Algorithms (WAOA)*. Wrocław, Poland, 2014, pp. 168–180. DOI: 10.1007/978-3-319-18263-6_15.

[ESS14b] Martina Eikel, Christian Scheideler, and Alexander Setzer. **RoBuSt: A Crash-Failure-Resistant Distributed Storage System**. In: *Proceedings of the 18th International Conference on Principles of Distributed Systems (OPODIS)*. Cortina d'Ampezzo, Italy, 2014, pp. 107–122. DOI: 10.1007/978-3-319-14472-6_8.

[FKS18] Michael Feldmann, Christina Kolb, and Christian Scheideler. **Self-stabilizing Overlays for High-Dimensional Monotonic Searchability**. In: *Proceedings of the 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Tokyo, Japan, 2018, pp. 16–31. DOI: 10.1007/978-3-030-03232-6_2.

[For+14] Dianne Foreback, Andreas Koutsopoulos, Mikhail Nesterenko, Christian Scheideler, and Thim Strothmann. **On Stabilizing Departures in Overlay Networks**. In: *Proceedings of the 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Paderborn, Germany, 2014, pp. 48–62. DOI: 10.1007/978-3-319-11764-5_4.

[FSS18] Michael Feldmann, Christian Scheideler, and Alexander Setzer. **Skueue: A Scalable and Sequentially Consistent Distributed Queue**. In: *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Vancouver, British Columbia, Canada, 2018, pp. 1040–1049. DOI: 10.1109/IPDPS.2018.00113.

[FSS20] Michael Feldmann, Christian Scheideler, and Stefan Schmid. **Survey on Algorithms for Self-Stabilizing Overlay Networks**. In: *ACM Computing Surveys* (2020). DOI: 10.1145/3397190.

[Gal+14] Dominik Gall, Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. **A Note on the Parallel Runtime of Self-Stabilizing Graph Linearization**. In: *Theory of Computing Systems* 55.1 (2014), pp. 110–135. DOI: 10.1007/s00224-013-9504-x.

[GK15] Anupam Gupta and Amit Kumar. **Greedy Algorithms for Steiner Forest**. In: *Proceedings of the 47th Annual ACM Symposium on Theory of Computing (STOC)*. Portland, Oregon, USA, 2015, pp. 871–878. DOI: 10.1145/2746539.2746590.

[Gro+18] Martin Groß, Anupam Gupta, Amit Kumar, Jannik Matuschke, Daniel R. Schmidt, Melanie Schmidt, and José Verschae. **A Local-Search Algorithm for Steiner Forest**. In: *Proceedings of the 9th Innovations in Theoretical Computer Science Conference (ITCS)*. Cambridge, Massachusetts, USA, 2018, pp. 31:1–31:17. DOI: 10.4230/LIPIcs.ITCS.2018.31.

[GSS18] Thorsten Götte, Christian Scheideler, and Alexander Setzer. **On Underlay-Aware Self-Stabilizing Overlay Networks**. In: *Proceedings of the 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Tokyo, Japan, 2018, pp. 50–64. DOI: 10.1007/978-3-030-03232-6_4.

[GW95] Michel X. Goemans and David P. Williamson. **A General Approximation Technique for Constrained Forest Problems**. In: *SIAM Journal on Computing* 24.2 (1995), pp. 296–317. DOI: 10.1137/S0097539793242618.

[Hec06] Reiko Heckel. **Graph Transformation in a Nutshell**. In: *Electronic Notes in Theoretical Computer Science* 148.1 (2006), pp. 187–198. DOI: 10.1016/j.entcs.2005.12.018.

[HFK06] Vincent C. Hu, David Ferraiolo, and D. Richard Kuhn. **Assessment of access control systems**. US Department of Commerce, National Institute of Standards and Technology, 2006.

[Hun+05] Galen Hunt, James R. Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, et al. *An overview of the Singularity project*. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[Jac+12] Riko Jacob, Stephan Ritscher, Christian Scheideler, and Stefan Schmid. **Towards higher-dimensional topological self-stabilization: A distributed algorithm for Delaunay graphs**. In: *Theoretical Computer Science* 457 (2012), pp. 137–148. DOI: 10.1016/j.tcs.2012.07.029.

[Jac+14] Riko Jacob, Andréa W. Richa, Christian Scheideler, Stefan Schmid, and Hanjo Täubig. **SKIP$^+$: A Self-Stabilizing Skip Graph**. In: *Journal of the ACM* 61.6 (2014), pp. 36:1–36:26. DOI: 10.1145/2629695.

[Jai01] Kamal Jain. **A Factor 2 Approximation Algorithm for the Generalized Steiner Network Problem**. In: *Combinatorica* 21.1 (2001), pp. 39–60. DOI: 10.1007/s004930170004.

[JM10] Colette Johnen and Fouzi Mekhaldi. **Robust Self-stabilizing Construction of Bounded Size Weight-Based Clusters**. In: *Proceedings (Part I) of the 16th European Conference on Parallel Processing (Euro-Par)*. Ischia, Italy, 2010, pp. 535–546. DOI: 10.1007/978-3-642-15277-1_51.

[Kar72] Richard M. Karp. **Reducibility Among Combinatorial Problems**. In: *Proceedings of a Symposium on the Complexity of Computer Computations*. New York, USA, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[KKS12] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. **A Self-Stabilization Process for Small-World Networks**. In: *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Shanghai, China, 2012, pp. 1261–1271. DOI: 10.1109/IPDPS.2012.115.

[KKS14] Sebastian Kniesburges, Andreas Koutsopoulos, and Christian Scheideler. **Re-Chord: A Self-stabilizing Chord Overlay Network**. In: *Theory of Computing Systems* 55.3 (2014), pp. 591–612. DOI: 10.1007/s00224-012-9431-2.

[KM05] Hervé Kerivin and Ali Ridha Mahjoub. **Design of Survivable Networks: A survey**. In: *Networks* 46.1 (2005), pp. 1–21. DOI: 10.1002/net.20072.

[KM06] Hirotsugu Kakugawa and Toshimitsu Masuzawa. **A self-stabilizing minimal dominating set algorithm with safe convergence**. In: *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*. Rhodes Island, Greece, 2006. DOI: 10.1109/IPDPS.2006.1639550.

[KMD17] Sabrina Kirrane, Alessandra Mileo, and Stefan Decker. **Access control and the Resource Description Framework: A survey**. In: *Semantic Web* 8.2 (2017), pp. 311–352. DOI: 10.3233/SW-160236.

[Kos09] Hristo Koshutanski. **A Survey on Distributed Access Control Systems for Web Business Processes**. In: *International Journal of Network Security* 9.1 (2009), pp. 61–69.

[Kru+10] Lachezar Krumov, Immanuel Schweizer, Dirk Bradler, and Thorsten Strufe. **Leveraging Network Motifs for the Adaptation of Structured Peer-to-Peer-Networks**. In: *Proceedings of the Global Communications Conference 2010 (GLOBECOM)*. Miami, Florida, USA, 2010, pp. 1–5. DOI: 10.1109/GLOCOM.2010.5683139.

[KSS17] Andreas Koutsopoulos, Christian Scheideler, and Thim Strothmann. **Towards a universal approach for the finite departure problem in overlay networks**. In: *Information and Computation* 255 (2017), pp. 408–424. DOI: 10.1016/j.ic.2016.12.006.

[Lev73] Leonid Anatolevich Levin. **Universal sequential search problems**. In: *Problemy Peredachi Informatsii* 9.3 (1973), pp. 115–116.

[Lin94] Chih-Long Lin. **Hardness of Approximating Graph Transformation Problem**. In: *Proceedings of the 5th International Symposium on Algorithms and Computation (ISAAC)*. Beijing, China, 1994, pp. 74–82. DOI: 10.1007/3-540-58325-4_168.

[Liu+06] Xiaomei Liu, Li Xiao, Andrew Kreling, and Yunhao Liu. **Optimizing overlay topology by reducing cut vertices**. In: *Proceedings of the 16th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*. Newport, Rhode Island, USA, 2006, pp. 17:1–17:6. DOI: 10.1145/1378191.1378213.

[LMM10] Aliaksandr Lazouski, Fabio Martinelli, and Paolo Mori. **Usage control in computer security: A survey**. In: *Computer Science Review* 4.2 (2010), pp. 81–99. DOI: 10.1016/j.cosrev.2010.02.002.

[LSS19] Linghui Luo, Christian Scheideler, and Thim Strothmann. **MULTISKIPGRAPH: A Self-Stabilizing Overlay Network that Maintains Monotonic Searchability**. In: *Proceedings of the 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Rio de Janeiro, Brazil, 2019, pp. 845–854. DOI: 10.1109/IPDPS.2019.00093.

[Mil+08] Stefan Miltchev, Jonathan M. Smith, Vassilis Prevelakis, Angelos D. Keromytis, and Sotiris Ioannidis. **Decentralized access control in distributed file systems**. In: *ACM Computing Surveys* 40.3 (2008), pp. 10:1–10:30. DOI: 10.1145/1380584.1380588.

[NNS13] Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. **Corona: A stabilizing deterministic message-passing skip list**. In: *Theoretical Computer Science* 512 (2013), pp. 119–129. DOI: 10.1016/j.tcs.2012.08.029.

[NNT13] Rizal Mohd Nor, Mikhail Nesterenko, and Sébastien Tixeuil. **Linearizing Peer-to-Peer Systems with Oracles**. In: *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Osaka, Japan, 2013, pp. 221–236. DOI: 10.1007/978-3-319-03089-0_16.

[ORS07] Melih Onus, Andréa W. Richa, and Christian Scheideler. **Linearization: Locally Self-Stabilizing Sorting in Graphs**. In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. New Orleans, Louisiana, USA, 2007, pp. 99–108. DOI: 10.1137/1.9781611972870.10.

[Roz97] Grzegorz Rozenberg, ed. **Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations**. World Scientific, 1997. DOI: 10.1142/3303.

[RSS11] Andréa W. Richa, Christian Scheideler, and Phillip Stevens. **Self-Stabilizing De Bruijn Networks**. In: *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Grenoble, France, 2011, pp. 416–430. DOI: 10.1007/978-3-642-24550-3_31.

[RSS18] Peter Robinson, Christian Scheideler, and Alexander Setzer. **Breaking the $\tilde{\Omega}(\sqrt{n})$ Barrier: Fast Consensus under a Late Adversary**. In: *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Vienna, Austria, 2018, pp. 173–182. DOI: 10.1145/3210377.3210399.

[SR05] Ayman Shaker and Douglas S. Reeves. **Self-Stabilizing Structured Ring Topology P2P Systems**. In: *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing (P2P)*. Konstanz, Germany, 2005, pp. 39–46. DOI: 10.1109/P2P.2005.34.

[SS18] Christian Scheideler and Alexander Setzer. **Relays: A New Approach for the Finite Departure Problem in Overlay Networks**. In: *Proceedings of the 20th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Tokyo, Japan, 2018, pp. 239–253. DOI: 10.1007/978-3-030-03232-6_16.

[SS19] Christian Scheideler and Alexander Setzer. **On the Complexity of Local Graph Transformations**. In: *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP)*. Patras, Greece, 2019, pp. 150:1–150:14. DOI: 10.4230/LIPIcs.ICALP.2019.150.

[SSS15] Christian Scheideler, Alexander Setzer, and Thim Strothmann. **Towards Establishing Monotonic Searchability in Self-Stabilizing Data Structures**. In: *Proceedings of the 19th International Conference on Principles of Distributed Systems (OPODIS)*. Rennes, France, 2015, pp. 24:1–24:17. DOI: 10.4230/LIPIcs.OPODIS.2015.24.

[SSS16] Christian Scheideler, Alexander Setzer, and Thim Strothmann. **Towards a Universal Approach for Monotonic Searchability in Self-stabilizing Overlay Networks**. In: *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*. Paris, France, 2016, pp. 71–84. DOI: 10.1007/978-3-662-53426-7_6.

[Ste+16] Michael Stein, Alexander Frömmgen, Roland Kluge, Frank Löffler, Andy Schürr, Alejandro P. Buchmann, and Max Mühlhäuser. **TARL: modeling topology adaptations for networking applications**. In: *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS@ICSE)*. Austin, Texas, USA, 2016, pp. 57–63. DOI: 10.1145/2897053.2897061.

[SWK69] K. Steiglitz, P. Weiner, and D. Kleitman. **The Design of Minimum-Cost Survivable Networks**. In: *IEEE Transactions on Circuit Theory* 16.4 (1969), pp. 455–460. DOI: 10.1109/TCT.1969.1083004.

[Wob+07] Ted Wobber, Aydan Yumerefendi, Martín Abadi, Andrew Birrell, and Daniel R. Simon. **Authorizing Applications in Singularity**. In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*. Lisbon, Portugal, 2007, pp. 355–368. DOI: 10.1145/1272996.1273033.

[YT10] Yukiko Yamauchi and Sébastien Tixeuil. **Monotonic Stabilization**. In: *Proceedings of the 14th International Conference on Principles of Distributed Systems (OPODIS)*. Tozeur, Tunisia, 2010, pp. 475–490. DOI: 10.1007/978-3-642-17653-1_34.